PERFORMANCE MODELING FRAMEWORK FOR SLO-DRIVEN MAPREDUCE ENVIRONMENTS

BY

ABHISHEK VERMA

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2012

Urbana, Illinois

Doctoral Committee:

Professor Roy H. Campbell, Chair
Professor Indranil Gupta
Professor William D. Gropp
Doctor Ludmila Cherkasova, HP Labs.

Abstract

Several companies are increasingly using MapReduce for efficient large scale data processing such as personalized advertising, spam detection, and data mining tasks. There is a growing need among MapReduce users to achieve different Service Level Objectives (SLOs). Often, applications need to complete data processing within a certain time deadline. Alternatively, users are interested in completing a set of jobs as fast as possible. Designing, prototyping, and evaluating new resource allocation and job scheduling algorithms to support these SLOs in MapReduce environments is challenging, labor-intensive, and time-consuming. Hence, accurate and efficient workload management and performance modeling tools are needed.

Our hypothesis is that *performance modeling of MapReduce environments through a combination of measurement, simulation, and analytical modeling for enabling different service level objectives is feasible, novel, and useful.* To support this hypothesis, we propose an analytical performance model based on key performance characteristics measured from past job executions and build a simulator capable of replaying these job traces. We survey different attempts at performance modeling and its applications, and contrast our work. To demonstrate the usefulness of our techniques, we apply them to achieve service level objectives such as enabling deadline-driven scheduling, optimizing makespan of a set of MapReduce jobs and comparing hardware alternatives.

To Papa, Mummy, Sonu, Soni and Moni.

Acknowledgments

This dissertation would not have been possible without the support of several people. Many thanks to my adviser, Prof. Roy H. Campbell and Dr. Ludmila Cherkasova, who gave me direction and taught me how to do research. I would like to acknowledge the important role my PhD committee for giving me feedback. And finally, thanks to my parents and numerous friends for always offering their love and support.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

We are living in the age of *Big Data*. There is a flood of data from several sources such as search engines, social networks and sensors. For example, Google processes more than 20 PB of data per day [1]. The New York Stock Exchange captures and stores about one terabyte of new trade data per day [2]. Facebook reports that they load between 10 - 15 TB of compressed (about 60 - 90 TB uncompressed) data every day [3]. The Internet Archive Wayback Machine contains almost 2 petabytes of data and is currently growing at a rate of 20 TB per month [4]. This eclipses the amount of text contained in the world's largest libraries, including the Library of Congress. The Large Hadron Collider produces about 15 PB of new data each year. Much more data is discarded during multi-level filtering before storage [5]. Thus, many enterprises, financial institutions and government organizations are experiencing a paradigm shift towards large-scale data intensive computing.

Analyzing large amount of unstructured data is a high priority task for many companies. The steep increase in the volume of information being produced often exceeds the capabilities of existing commercial databases. This is driving interest in alternatives such as MapReduce [6], Dryad [7], DryadLINQ [8], and Spark [9] for dealing with these requirements. MapReduce and its open-source implementation Hadoop present a new, popular choice: it offers an efficient distributed computing platform capable of handling large volumes of data and mining petabytes of unstructured information. To exploit and tame the power of information explosion, several companies [10] are piloting the use of Hadoop for large scale data processing. Hadoop is increasingly being used across enterprises for advanced data analytics and enabling new applications associated with data retention, regulatory compliance, e-discovery and litigation issues.

There is a growing need among MapReduce users to achieve different ser-

vice level objectives (SLOs): Users want the capability to estimate the completion time of their jobs. Also, they would like to determine the amount of resources that should be leased from a cloud computing provider or a shared cluster targeting specific performance goal. For example, Facebook insights [11] would like to perform real-time analytics across websites with social plug-ins, Facebook Pages, and Facebook Ads. Yahoo! [12] "would like to move to a model that is based on (soft) deadlines rather than the low-level mechanism of bounding the queue time. However, a deadline-based model hinging on the ability to model Pig/Hadoop execution times in a multi-tenant environment, is elusive". Datacenter machines typically get replaced over a period of three to five years [13]. Users would like to estimate the performance of their workloads on new hardware for capacity planning. Designing, prototyping, and evaluating new resource allocation and job scheduling algorithms in shared MapReduce environments to support these SLOs is a challenging, labor-intensive, and time-consuming task. Hence, accurate and efficient workload management and performance modeling tools are needed.

1.1 Hypothesis

Our hypothesis is that *performance modeling of MapReduce environments through a combination of measurement, simulation, and analytical modeling for enabling different service level objectives is feasible, novel, and useful.*

We support this hypothesis in three parts as follows:

1. **Feasibility:** We create an analytical MapReduce performance model based on key performance characteristics measured from past job executions and build a simulator capable of replaying these job traces.

2. **Novelty:** We survey different attempts at performance modeling and its applications, and contrast our work.

3. **Usefulness:** To demonstrate the usefulness of our models, we apply them to achieve different service level objectives such as enabling deadline-driven scheduling, optimizing makespan for a set of MapReduce jobs and comparing different hardware alternatives.

1.2 Intellectual Merit

Our work provides several algorithmic contributions and models for estimating the performance of MapReduce applications. First, we theoretically prove the upper and lower bounds on completion times for a set of tasks and use it to develop a practical, analytical model for estimating the completion time of MapReduce applications. Second, we solve the problem of determining the minimum resources that should should be allocated to the job so that it completes within a given deadline. Third, we create a fast and accurate simulator capable of replaying MapReduce traces efficiently. This enables experimenting with different scheduling policies and workload management techniques. Finally, our performance models help in evaluating different hardware alternatives and planning the capacity of future clusters.

1.3 Broader Impact

Our performance model can be extended to more general data-intensive frameworks such as Dryad [7], Pig [14], and Hive [15]. Our performance modeling techniques enable the prediction of performance of current data-intensive computing applications and make them faster. They empower the cloud users by allowing them to achieve different service level objectives

(SLOs), help in meeting deadline requirements, and answering *what-if* questions. Our simulation framework assists cloud providers in experimenting with different scheduling policies and help them avoid error-prone, guess-based decisions. By allocating resources in a systematic and efficient way, our systems optimize the total usage of the cloud resources.

1.4 Assumptions and Limitations

"All models are wrong. Some models are useful." – George Box.
We make three assumptions about the MapReduce environment being modeled. First, we assume that the completion time of map and reduce tasks depends only on the total amount of data and not its actual contents. This is typical of most MapReduce applications [6]. For example, we would not be able to accurately predict the completion time of a MapReduce program that tries to factorize numbers, since the runtime would depend on the actual number being factorized. Second, we do not attempt to model the network and assume that it is not a bottleneck in the system. This assumption is validated with the advent of new, high bisection bandwidth datacenter topologies such as Fat tree [16], VL2 [17] and BCube [18]. Finally, we assume that there is no background load on the MapReduce clusters.

1.5 Summary of Contributions

The main contributions of this research are (1) the design of a performance model through a combination of measurement, simulation, and analysis; and (2) the application of these techniques for achieving different service level objectives.

Our performance model enables estimating the job completion time and solving the inverse problem of finding the appropriate amount of resources for a job that needs to meet a given (soft) deadline. We use a combination of measurement, simulation and analytical modeling. We profile the past execution of the job and extract a compact form of the key performance indicators of the job. We approximate the completion time of a MapReduce job by estimating the lower and upper bounds of the job execution time as a function of the input dataset size and allocated resources. From this closed form expression and using the Lagrange's multipliers method, we determine the minimum amount of resources that should be allocated in order to complete the job within a given deadline.

Instead of profiling the job by executing on the entire dataset, we can profile the job by executing it on a few small input datasets and extrapolating its performance profile. We propose and validate linear regression for estimating the duration of the shuffle and reduce phases separately. We model the effects of failures during the execution of the job, when the failed resources are replenished and in the case they cannot be replenished. To enable users and administrators to experiment with different scheduling policies, set different pool sizes and assign job priorities, we develop a fast and accurate MapReduce simulator called SimMR. At the heart of SimMR is a discrete event engine that is capable of replaying real and synthetically generated job traces.

We apply these performance modeling tools for achieving different service level objectives: First, we use our analytical model to determine the set of plausible resource allocations to satisfy a soft deadline provided by the user for a single MapReduce job. Extending this idea for a set of MapReduce jobs, we propose a novel scheduler which incorporates mechanisms for job ordering, resource provisioning and allocation/deallocation of spare resources. Second, we solve the resource provisioning problem for Pig programs, which translate to a directed acyclic graph (DAG) of MapReduce jobs. Third, we discuss

the problem of minimizing the Makespan of a set of MapReduce jobs. Finally, we discuss the problem of estimating the performance of MapReduce applications on different hardware alternatives.

1.6 Dissertation Outline

The dissertation is structured as follows: Chapter 2 provides a brief overview of the MapReduce abstraction, execution and failure modes. Chapter 3 introduces ARIA – Automated Resource Inference and Allocation for deadline-driven scheduling. Chapter 4 discusses the design and implementation of a fast discrete event simulator SimMR for MapReduce jobs. Chapter 5 discusses techniques for improving makespan of a set of independent MapReduce jobs. Chapter 6 introduces additional modeling techniques to compare different hardware alternatives for MapReduce workloads. Chapter 7 gives an overview of the state of the art and contrasts our work. Finally, Chapter 8 summarizes our work and concludes the thesis.

Chapter 2

MapReduce Background

This section provides an overview of the MapReduce [6] abstraction, execution, scheduling, and failure modes. In the MapReduce model, computation is expressed as two functions: map and reduce. The map function takes an input key/value pair and produces a list of intermediate key/value pairs. The intermediate values associated with the same key k_2 are grouped together and then passed to the reduce function. The reduce function takes intermediate key k_2 with a list of values and processes them to form a new list of values.

$$\begin{aligned} map(k_1, v_1) &\rightarrow list(k_2, v_2) \\ reduce(k_2, list(v_2)) &\rightarrow list(v_3) \end{aligned}$$

MapReduce jobs are distributed and executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

Each map task processes a logical split of input data that generally resides on a distributed file system. Files are typically divided into uniform sized blocks and distributed across the cluster nodes. The map task reads data, applies the user-defined map function on each record, and buffers the resulting output. This data is sorted and partitioned for different reduce tasks, and written to the local disk of the machine executing the map task.

The reduce stage consists of two phases: *shuffle* and *reduce* phase. In the *shuffle phase*, the reduce tasks fetch the intermediate data files from the already completed map tasks. The intermediate files from all the map tasks are sorted. An external merge sort is used in case the intermediate data does not fit in memory as follows: the intermediate data is shuffled, merged in memory, and written to disk. After all the intermediate data is shuffled, a final pass is made to merge all these sorted files. Thus, the shuffling and sorting of intermediate is interleaved: we denote this by *shuffle/sort* or simply

shuffle phase. Finally, in the *reduce phase*, the sorted intermediate data is passed to the user-defined reduce function. The output from the reduce function is generally written back to the distributed file system.

Job scheduling in Hadoop is performed by a master node, which manages a number of worker nodes in the cluster. Each worker has a fixed number of *map slots* and *reduce slots*, which can run tasks. The number of map and reduce slots is statically configured. The slaves periodically send heartbeats to the master to report the number of free slots and the progress of tasks that they are currently running. Based on the availability of free slots and the scheduling policy, the master assigns map and reduce tasks to slots in the cluster.

In the real world, user code is buggy, processes crash, and machines fail. MapReduce is designed to scale to a large number of machines and to yield a graceful performance degradation in case of failures. There are three types of failures that can occur. First, a map or reduce task can fail because of buggy code or runtime exceptions. The worker node running the failed task detects task failures and notifies the master. The master reschedules the execution of the failed task, preferably on a different machine. Secondly, a worker can fail, e.g., because of OS crash, faulty hard disk, or network interface failure. The master notices a worker that has not sent any heartbeats for a specified time interval and removes it from its worker pool for scheduling new tasks. Any tasks in progress on the failed worker are rescheduled for execution. Finally, the failure of the master is the most serious failure mode. This failure is rare and can be avoided by running multiple masters and using the Paxos [19] consensus protocol to decide the primary master.

Chapter 3

Automated Resource Inference and Allocation

3.1 Motivation

Many enterprises, financial institutions and government organizations are experiencing a paradigm shift towards large-scale data intensive computing. Analyzing large amount of unstructured data is a high priority task for many companies. The steep increase in volume of information being produced often exceeds the capabilities of existing commercial databases. Moreover, the performance of physical storage is not keeping up with improvements in network speeds. All these factors are driving interest in alternatives that can propose a better paradigm for dealing with these requirements. MapReduce [6] and its open-source implementation Hadoop present a new, popular alternative: it offers an efficient distributed computing platform for handling large volumes of data and mining petabytes of unstructured information. To exploit and tame the power of information explosion, many companies[1] are piloting the use of Hadoop for large scale data processing. It is increasingly being used across the enterprise for advanced data analytics and enabling new applications associated with data retention, regulatory compliance, e-discovery and litigation issues.

In the enterprise setting, users would benefit from sharing Hadoop clusters and consolidating diverse applications over the same datasets. Originally, Hadoop employed a simple FIFO scheduling policy. Designed with a primary goal of minimizing the makespan of large, routinely executed batch workloads, the simple *FIFO* scheduler is quite efficient. However, job management using this policy is very inflexible: once long, production jobs are scheduled in the MapReduce cluster the later submitted short, interactive ad-hoc queries must wait until the earlier jobs finish, which can make their

[1] "Powered by" Hadoop, http://wiki.apache.org/hadoop/PoweredBy

outcomes less relevant. The Hadoop Fair Scheduler (HFS) [20] solves this problem by enforcing some fairness among the jobs and guaranteeing that each job at least gets a predefined minimum of allocated slots. While this approach allows sharing the cluster among multiple users and their applications, HFS does not provide any support or control of allocated resources in order to achieve the application performance goals and service level objectives (SLOs).

In MapReduce environments, many production jobs are run periodically on new data. For example, Facebook, Yahoo!, and eBay process terabytes of data and event logs per day on their Hadoop clusters for spam detection, business intelligence and different types of optimization. For users who require service guarantees, a performance question to be answered is the following: given a MapReduce job J with input dataset D, how many map/reduce slots need to be allocated to this job over time so that it finishes within (soft) deadline T?

Currently, there is no job scheduler for MapReduce environments that given a job completion deadline, could estimate and allocate the appropriate number of map and reduce slots to the job so that it meets the required deadline. In this work, we design a framework, called **ARIA** (**A**utomated **R**esource **I**nference and **A**llocation), to address this problem. It is based on three inter-related components.

- For a production job that is routinely executed on a new dataset, we build a *job profile* that reflects critical performance characteristics of the underlying application during map, shuffle, sort, and reduce phases.

- We design a *MapReduce performance model*, that for a given job (with a known profile), the amount of input data for processing and a specified soft deadline (job's SLO), estimates the amount of map and reduce slots required for the job completion within the deadline.

- We implement a novel *SLO-scheduler* in Hadoop that determines job ordering and the amount of resources that need to be allocated for meeting the job's SLOs. The job ordering is based on the EDF policy (*Earliest Deadline First*). For resource allocation, the new scheduler relies on the designed performance model to suggest the appropriate number of map and reduce slots for meeting the job deadlines. The

resource allocations are dynamically recomputed during the job's execution and adjusted if necessary.

We validate our approach using a diverse set of realistic applications. The application profiles are stable and the predicted completion times are within 15% of the measured times in the testbed. The new scheduler effectively meets the jobs' SLOs until the job demands exceed the cluster resources. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster[2].

This chapter is organized as follows. Section 3.2 introduces profiling of MapReduce jobs. Section 3.3 establishes performance bounds on job completion time. The initial evaluation of introduced concepts is done in Section 3.4. We discuss how we can profile jobs on a smaller dataset and extrapolate their performance in Section 3.5 and discuss the impact of failures in Section 3.6. We design an SLO-based performance model for MapReduce jobs in Section 3.7. Section 3.8 outlines the ARIA implementation. We evaluate the efficiency of the new scheduler in Section 3.9. Finally, we summarize the results in Section 3.10.

3.2 Job Executions and Job Profile

The amount of allocated resources may drastically impact the job progress over time. In this section, we discuss different executions of the same MapReduce job as a function of the allocated map and reduce slots. Our goal is to extract a single *job profile* that uniquely captures critical performance characteristics of the job execution in different stages.

3.2.1 Job Executions

MapReduce jobs are distributed and executed across multiple machines: the map stage is partitioned into *map tasks* and the reduce stage is partitioned into *reduce tasks*.

Let us demonstrate different executions of the same job using the *sort benchmark* [21], which involves the use of identity map/reduce function (i.e.,

[2]It is a representative cluster size for many enterprises. The Hadoop World 2010 survey reported the average cluster size as 66 nodes.

the entire input of map tasks is shuffled to reduce tasks and written as output). First, we run the sort benchmark with 8GB input on 64 machines, each configured with a single map and a single reduce slot, i.e., with 64 map and 64 reduce slots overall.

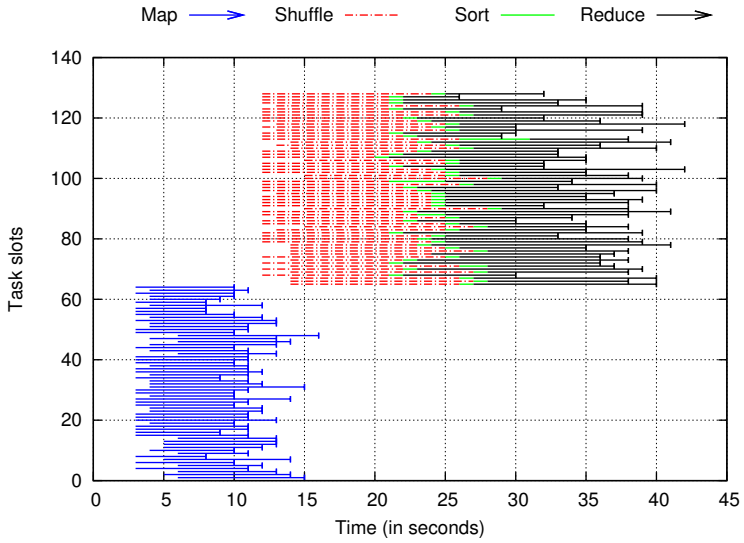


Figure 3.1: Sorting with 64 map and 64 reduce slots.

Figure 3.1 shows the progress of the map and reduce tasks over time (on the x-axis) vs the 64 map slots and 64 reduce slots (on the y-axis). Since the file blocksize is 128MB, there are 8GB/128MB = 64 input splits. As each split is processed by a different map task, the job consists of 64 map tasks. This job execution results in single map and reduce *wave*. We split each reduce task into its constituent shuffle, sort and reduce phases (we show the sort phase duration that is complementary to the shuffle phase). As seen in the figure, a part of the shuffle phase overlaps with the map stage.

Next, we run the sort benchmark with 8GB input on the same testbed, except this time, we provide it with a fewer resources: 16 map slots and

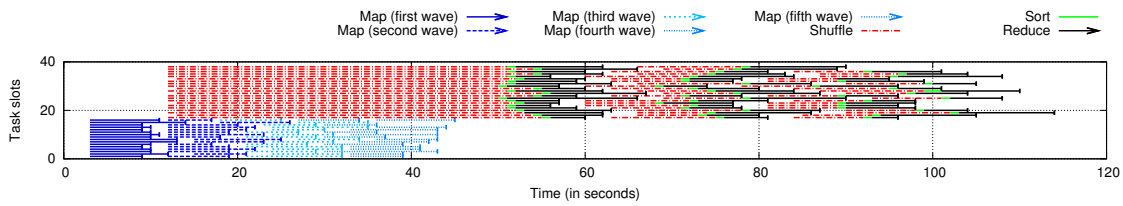


Figure 3.2: Sorting with 16 map and 22 reduce slots.

22 reduce slots. As shown in Figure 3.2, since the number of map tasks is greater than the number of provided map slots, the map stage proceeds in multiple rounds of slot assignment, viz. 4 waves ($\lceil 64/16 \rceil$) and the reduce stage proceeds in 3 waves ($\lceil 64/22 \rceil$).

As observed from Figures 3.1 and 3.2, it is difficult to predict the completion time of the same job when different amount of resources are given to the job. In the next section, we introduce a job profile that can be used for prediction of the job completion time as a function of assigned resources.

3.2.2 Job Profile

Our goal is to create a compact job profile that is comprised of performance *invariants* which are independent on the amount of resources assigned to the job over time and that reflects all phases of a given job: map, shuffle, sort, and reduce phases. This information can be obtained from the counters at the job master during the job's execution or parsed from the logs. More details can be found in Section 3.8.

The **map stage** consists of a number of map tasks. To compactly characterize the task duration distribution and other invariant properties, we extract the following metrics:
$(M_{min}, M_{avg}, M_{max}, AvgSize_M^{input}, Selectivity_M)$, where

- M_{min} – the minimum map task duration. M_{min} serves as an estimate for the beginning of the shuffle phase since it starts when the first map task completes.
- M_{avg} – the average duration of map tasks to summarize the duration of a map wave.
- M_{max} – the maximum duration of map tasks. It is used as a worst time estimate for a map wave completion.
- $AvgSize_M^{input}$ - the average amount of input data per map task. We use it to estimate the number of map tasks to be spawned for processing a new dataset.
- $Selectivity_M$ – the ratio of the map output size to the map input size. It is used to estimate the amount of intermediate data produced by the

map stage.

As described earlier, the **reduce stage** consists of the shuffle/sort and reduce phases.

The **shuffle/sort phase** begins only after the first map task has completed. The shuffle phase (of any reduce wave) completes when the entire map stage is complete and all the intermediate data generated by the map tasks has been shuffled to the reduce tasks and has been sorted. Since the shuffle and sort phases are interleaved, we do not consider the sort phase separately and include it in the shuffle phase. After shuffle/sort completes, the reduce phase is performed. Thus the profiles of shuffle and reduce phases are represented by the *average* and *maximum* of their tasks durations. Note, that the measured shuffle durations include networking latencies for the data transfers that reflect typical networking delays and contention specific to the cluster.

The shuffle phase of the first reduce wave may be significantly different from the shuffle phase that belongs to the next reduce waves (illustrated in Figure 3.2). This happens because the shuffle phase of the first reduce wave overlaps with the entire map stage and depends on the number of map waves and their durations. Therefore, we collect two sets of measurements: (Sh^1_{avg}, Sh^1_{max}) for shuffle phase of the first reduce wave (called, *first shuffle*) and $(Sh^{typ}_{avg}, Sh^{typ}_{max})$ for shuffle phase of the other waves (called, *typical shuffle*). Since we are looking for the performance invariants that are independent of the amount of allocated resources to the job, we characterize a shuffle phase of the first reduce wave in a special way and include only the non-overlapping portions of the first shuffle in (Sh^1_{avg} and Sh^1_{max}). Thus the job profile in the shuffle phase is characterized by two pairs of measurements: $(Sh^1_{avg}, Sh^1_{max}, Sh^{typ}_{avg}, Sh^{typ}_{max})$.

The **reduce phase** begins only after the shuffle phase is complete. The profile of the reduce phase is represented by: $(R_{avg}, R_{max}, Selectivity_R)$: the *average* and *maximum* of the reduce tasks durations and the *reduce selectivity*, denoted as $Selectivity_R$, which is defined as the ratio of the reduce output size to its input.

3.3 Estimating Job Completion Time

In this section, we design a MapReduce performance model that is based on *i)* the job profile and *ii)* the performance bounds of completion time of different job phases. This model can be used for predicting the job completion time as a function of the input dataset size and allocated resources.

3.3.1 Theoretical Bounds

First, we establish the performance bounds for a makespan (a completion time) of a given set of n tasks that is processed by k servers (or by k slots in MapReduce environments).

Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks of a given job. Let k be the number of slots that can each execute one task at a time. The assignment of tasks to slots is done using a simple, online, *greedy* algorithm, i.e., assign each task to the slot with the earliest finishing time.

Let $\mu = (\sum_{i=1}^{n} T_i)/n$ and $\lambda = \max_i \{T_i\}$ be the *mean* and *maximum duration* of the n tasks respectively.

Makespan Theorem: The makespan of the greedy task assignment is at least $n \cdot \mu/k$ and at most $(n-1) \cdot \mu/k + \lambda$. [3]

The lower bound is trivial, as the best case is when all n tasks are equally distributed among the k slots (or the overall amount of work $n \cdot \mu$ is processed as fast as possible by k slots). Thus, the overall makespan is at least $n \cdot \mu/k$.

For the upper bound, let us consider the worst case scenario, i.e., the longest task $\hat{T} \in \{T_1, T_2, \ldots, T_n\}$ with duration λ is the last processed task. In this case, the time elapsed before the final task $\hat{T}$ is scheduled is at most the following: $(\sum_{i=1}^{n-1} T_i)/k \leq (n-1) \cdot \mu/k$. Thus, the makespan of the overall assignment is at most $(n-1) \cdot \mu/k + \lambda$. [4]■

The difference between lower and upper bounds represents the range of possible job completion times due to non-determinism and scheduling. These bounds are particularly useful when $\lambda \ll n \cdot \mu/k$, i.e., when the duration of

[3] Tighter lower and upper bounds can be defined for some special cases, e.g., if $n \leq k$ then lower and upper bounds are equal to λ, or lower bound can be defined as $max(n \cdot \mu/k, \lambda)$. However, this would complicate the general computation. Typically, for multiple waves, the proposed bounds are tight. Since our MapReduce model actively uses Makespan Theorem, we chose to use a simpler version of the lower and upper bounds.

[4] Similar ideas were explored in the classic papers on scheduling, e.g., to characterize makespan bounds [22].

the longest task is small as compared to the total makespan.

3.3.2 Completion Time Estimates of a MapReduce Job

Let us consider job J with a given profile either built from executing this job in a staging environment or extracted from past job executions. Let J be executed with a new dataset that is partitioned into N_M^J map tasks and N_R^J reduce tasks. Let S_M^J and S_R^J be the number of map and reduce slots allocated to job J respectively.

Let M_{avg} and M_{max} be the average and maximum durations of map tasks (defined by the job J profile). Then, by Makespan Theorem, the lower and upper bounds on the duration of the entire map stage (denoted as T_M^{low} and T_M^{up} respectively) are estimated as follows:

$$T_M^{low} = N_M^J \cdot M_{avg} / S_M^J \tag{3.1}$$

$$T_M^{up} = (N_M^J - 1) \cdot M_{avg} / S_M^J + M_{max} \tag{3.2}$$

The reduce stage consists of shuffle (which includes the interleaved sort phase) and reduce phases. Similarly, Makespan Theorem can be directly applied to compute the lower and upper bounds of completion times for reduce phase (T_R^{low}, T_R^{up}) since we have measurements for average and maximum task durations in the reduce phase, the numbers of reduce tasks N_R^J and allocated reduce slots S_R^J. [5]

The subtlety lies in estimating the duration of the shuffle phase. We distinguish the non-overlapping portion of the *first shuffle* and the task durations in the *typical shuffle* (see Section 6.3 for definitions). The portion of the typical shuffle phase in the remaining reduce waves is computed as follows:

$$T_{Sh}^{low} = \left(\frac{N_R^J}{S_R^J} - 1 \right) \cdot Sh_{avg}^{typ} \tag{3.3}$$

$$T_{Sh}^{up} = \left(\frac{N_R^J - 1}{S_R^J} - 1 \right) \cdot Sh_{avg}^{typ} + Sh_{max}^{typ} \tag{3.4}$$

Finally, we can put together the formulae for the lower and upper bounds of

[5] For simplicity of explanation, we omit the normalization step of measured durations in job profile with respect to $AvgSize_M^{input}$ and $Selectivity_M$.

the overall completion time of job J:

$$T_J^{low} = T_M^{low} + Sh_{avg}^1 + T_{Sh}^{low} + T_R^{low} \tag{3.5}$$

$$T_J^{up} = T_M^{up} + Sh_{max}^1 + T_{Sh}^{up} + T_R^{up} \tag{3.6}$$

T_J^{low} and T_J^{up} represent optimistic and pessimistic predictions of the job J completion time. In Section 3.4, we compare whether the prediction that is based on the average value between the lower and upper bounds tends to be closer to the measured duration. Therefore, we define:

$$T_J^{avg} = (T_M^{up} + T_J^{low})/2. \tag{3.7}$$

Note that we can re-write Eq. 3.5 for T_J^{low} by replacing its parts with more detailed Eq. 3.1 and Eq. 3.3 and similar equations for sort and reduce phases as it is shown below:

$$T_J^{low} = \frac{N_M^J \cdot M_{avg}}{S_M^J} + \frac{N_R^J \cdot (Sh_{avg}^{typ} + R_{avg})}{S_R^J} + Sh_{avg}^1 - Sh_{avg}^{typ} \tag{3.8}$$

This presentation allows us to express the estimates for completion time in a simplified form shown below:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low}, \tag{3.9}$$

where $A_J^{low} = M_{avg}$, $B_J^{low} = (Sh_{avg}^{typ} + R_{avg})$, and $C_J^{low} = Sh_{avg}^1 - Sh_{avg}^{typ}$. Eq. 4.1 provides an explicit expression of a job completion time as a function of map and reduce slots allocated to job J for processing its map and reduce tasks, i.e., as a function of (N_M^J, N_R^J) and (S_M^J, S_R^J).
The equations for T_J^{up} and T_J^{avg} can be written similarly.

3.4 Initial Evaluation of Approach

In this section, we perform a set of initial performance experiments to justify and validate the proposed modeling approach based on application profiling. We use a motivating example WikiTrends for these experiments and later evaluate five other applications in Section 3.9.

3.4.1 Experimental Testbed

We perform our experiments on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39MHz cores, 8 GB RAM and two 160GB hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. This is a well-balanced configuration with disk I/O being a primary bottleneck. We used Hadoop 0.20.2 with two machines as the JobTracker and the NameNode, and remaining 64 machines as workers. Each worker is configured with four map and four reduce slots (unless explicitly specified otherwise). The file system blocksize is set to 64MB. The replication level is set to 3. Speculative execution is disabled as it did not lead to significant improvements in our experiments.

In order to validate our model, we use the data from the Trending Topics (TT)[6]: Wikipedia article traffic logs that were collected (and compressed) every hour in September and October 2010. We group these hourly logs according to the month. Our MapReduce application, called *WikiTrends*, counts the number of times each article has been visited according to the given input dataset, which is very similar to the job that is run periodically by TT.

3.4.2 Stability of Job Profiles

In our first set of experiments, we investigate whether the job profiles for a given job are stable across different input datasets and across different executions on the same dataset. To this end, we execute our MapReduce job on the September and October datasets and with variable number of map and reduce slots. The job profiles for the map and reduce stage are summarized in Table 3.1. The job "Month$_{x,y}$" denotes the MapReduce job run on the logs of the given month with x number of map slots and y number of reduce slots allocated to it. Table 3.1 shows that the map stage of the job profile is stable across different job executions and different datasets used as input data.

Table 3.1 also shows the job profiles for tasks in shuffle/sort and reduce phases. The shuffle statistics include two sets of data: the average and maximum duration of the non-overlapping portion of the first and typical

[6] http://trendingtopics.org

Job	Map Task duration			Avg Input Size in MB	Selectivity
	Min	Avg	Max		
$\text{Sept}_{256,256}$	94	144	186	59.85	10.07
$\text{Oct}_{256,256}$	86	142	193	58.44	9.98
$\text{Sept}_{64,128}$	94	133	170	59.85	10.07
$\text{Oct}_{64,128}$	71	132	171	58.44	9.98

Job	Shuffle/Sort		Reduce		
	Avg	Max	Avg	Max	Selectivity
$\text{Sept}_{256,256}$	12	20	16	33	0.37
	121	152			
$\text{Oct}_{256,256}$	13	34	17	35	0.36
	122	156			
$\text{Sept}_{64,128}$	10	25	15	139	0.37
	122	152			
$\text{Oct}_{64,128}$	11	26	15	109	0.36
	123	153			

Table 3.1: Map and Reduce profiles of four jobs.

shuffle waves. We performed the experiment 10 times and observed less than 5% variation. The average metric values are very consistent across different job instances (these values are most critical for our model). The maximum values show more variance. To avoid the outliers and to improve the robustness of the measured maximum durations we can use instead the mean of a few top values. From these measurements, we conclude that job profiles across different input datasets and across different executions of the same job are indeed similar.

3.4.3 Prediction of Job Completion Times

In our second set of experiments, we try to predict the job completion times when the job is executed on a different dataset and with different numbers of map and reduce slots.

We build a job profile from the job executed on the September logs with 256 map and 256 reduce slots. Using this job profile and applying formulae described in Section 6.4, we predict job completion times of the following job configurations:

Figure 3.3: Comparison of predicted and measured job completion times across different job executions and datasets.

- September logs: 64 map and 128 reduce slots;
- October logs: 256 map and 256 reduce slots;
- October logs: 64 map and 128 reduce slots.

The results are summarized in Figure 3.8. We observe that the relative error between the predicted average time T_J^{avg} and the measured job completion time is less than **10%** in all these cases, and hence, the predictions based on T_J^{avg} are well suited for ensuring the job SLOs.

3.5 Scaling Factors

In the previous section, we showed how to extract the job profile and use it for predicting job completion time when different amounts of resources are used. When the job is executed on a larger dataset the number of map tasks and reduce tasks may be scaled proportionally if the application structure allows it. In some cases, the number of reduce tasks is statically defined, e.g., 24 hours a day, or the number of categories (topics) in Wikipedia, etc. When the job is executed on a larger dataset while the number of reduce tasks is kept constant, the durations of the reduce tasks naturally increase as the size of the intermediate data processed by each reduce task increases. The duration of the map tasks is not impacted because this larger dataset is split into a larger number of map tasks but each map task processes a similar portion of data. The natural attempt might be to derive a single scaling factor for reduce task duration as a function of the amount of processed data, and then use it for

the shuffle and reduce phase duration scaling as well. However, this might lead to inaccurate results. The reason is that the shuffle phase performs data transfer and its duration is mainly defined by the network performance. The reduce phase duration is defined by the application specific computation of the user supplied reduce function and significantly depends on the disk write performance. Thus, the duration scaling in these phases might be different. Consequently, we derive two scaling factors for shuffle and reduce phases separately, each one as a function of the processed dataset size.

Therefore in the staging environment, we perform a set of k experiments ($i = 1, 2, ..., k$) with a given MapReduce job for processing different size input datasets (while keeping the number of reduce tasks constant), and collect the job profile measurements. We derive scaling factors with linear regression in the following way. Let D_i be the amount of intermediate data for processing per reduce task, and let $Sh^{typ}_{i,avg}$ and $R_{i,avg}$ be the job profile measurements for shuffle and reduce phases respectively. Then, using linear regression, we solve the following sets of equations:

$$C_0^{Sh} + C_1^{Sh} \cdot D_i = Sh^{typ}_{i,avg}, (i = 1, 2, \cdots, k) \tag{3.10}$$

$$C_0^{R} + C_1^{R} \cdot D_i = R_{i,avg}, (i = 1, 2, \cdots, k) \tag{3.11}$$

Derived scaling factors (C_0^{Sh}, C_1^{Sh}) for shuffle phase and (C_0^R, C_1^R) for reduce phase are incorporated in the job profile. When job J processes an input dataset that leads to a different amount of intermediate data D_{new} per reduce task, its profile is updated as $Sh^{typ}_{avg} = C_0^{Sh} + C_1^{Sh} \cdot D_{new}$ and $R_{avg} = C_0^R + C_1^R \cdot D_{new}$. Similar scaling factors can be derived for maximum durations Sh^{typ}_{max} and R_{max} as well as for the first shuffle phase measurements.

3.6 Impact of Failures on the Completion Time Bounds

The performance implications of failures depend on the type of failures. For example, *disk failures* are typical, but their performance implications for running MapReduce jobs are very mild. It is because by default, each piece of data is replicated three times, and data that resides on a failed disk can be fetched from other locations. Moreover, for each data block with a number

of copies less than the default replication level, Hadoop will reconstruct the additional copies.

Worker failure is another typical type of failure, and its performance implications for a MapReduce job can be more serious. If the failure happens while the job was running, and the failed worker has either completed or in-progress job map tasks then all these map tasks need to be recomputed, since the intermediate data of these tasks might be unavailable to current or future reduce tasks. The same applies to the reduce tasks which were in progress on the failed worker: they need to be restarted on a different node.

Moreover, in order to understand the performance impact of a worker failure on job completion time, we need to consider not only *when* the failure happened, but also whether additional resources in the system can be allocated to the job to compensate for the failed worker. For example, if a worker failure happens in the very beginning of the map stage and the resources of the failed worker are immediately replenished with additional ones, then the lower and upper bounds of job completion time remain practically the same. However, if the failed worker resources are not replenished then the performance bounds are higher.

On the other hand, if a worker failure happens during the job's last wave of reduce tasks then all the completed map tasks that reside on the failed node as well as the reduce tasks that were in-progress on this node have to be re-executed, and even if the resources of the failed node are immediately replenished there are serious performance implications of this failure so late during the job execution. The latency for recomputing the map and reduce tasks of the failed node can not be hidden: this computation time is explicitly on the critical path of the job execution and is equivalent of adding entire map and reduce stage latency: $M_{max} + Sh^{typ}_{max} + R_{max}$.

Given the time of failure t_f, we try to quantify the job completion time bounds. Let us consider job J with a given profile, which is partitioned into N^J_M map tasks and N^J_R reduce tasks. Let the worker failure happen at some point of time t_f. There are two possibilities for the job J execution status at the time of failure, it is either in the map or the reduce stage. We can predict whether the failure happened during the map or reduce stage based on either using low or upper bounds of a completion time (or its average). Let us consider the computation based on the lower bound. We now describe how to approximate the number of map and reduce tasks yet to be completed

in both the cases.

- *Case (1):* Let us assume that the job execution is in the map stage at time t_f, i.e., $t_f \leq T_M^{low}$. In order to determine the number of map tasks yet to be processed, we approximate the number of completed ($N^J_{M_{done}}$) and failed ($N^J_{M_{fail}}$) tasks as follows:

$$N^J_{M_{done}} \cdot M_{avg}/S^J_M = t_f \implies N^J_{M_{done}} = \lfloor t_f \cdot S^J_M/M_{avg} \rfloor$$

If there are W worker nodes in the Hadoop cluster for job J processing and one of them fails, then

$$N^J_{M_{fail}} = \lfloor N^J_{M_{done}}/W \rfloor$$

Thus, the number of map and reduce tasks yet to be processed at time t_f (denoted as N^J_{M,t_f} and N^J_{R,t_f}) are determined as follows:

$$N^J_{M,t_f} = N^J_M - N^J_{M_{done}} + N^J_{M_{fail}} \text{ and } N^J_{R,t_f} = N^J_R$$

- *Case (2):* Let us now assume that the map stage is complete, and the job execution is in the reduce stage at time t_f, $t_f \geq T_M^{low}$ and all the map tasks N^J_M are completed. The number of completed reduce tasks $N^J_{R_{done}}$ at time t_f can be evaluated using Eq. 4.1:

$$B_J^{low} \cdot \frac{N^J_{R_{done}}}{S^J_R} = t_f - C_J^{low} - A_J^{low} \cdot \frac{N^J_M}{S^J_M}$$

Then the number of failed map and reduce tasks can be approximated as:

$$N^J_{M_{fail}} = \lfloor N^J_M/W \rfloor \text{ and } N^J_{R_{fail}} = \lfloor N^J_{R_{done}}/W \rfloor$$

The remaining map and reduce tasks of job J yet to be processed at time t_f are determined as follows:

$$N^J_{M,t_f} = N^J_{M_{fail}} \text{ and } N^J_{R,t_f} = N^J_R - N^J_{R_{done}} + N^J_{R_{fail}}$$

Let S^J_{M,t_f} and S^J_{R,t_f} be the number of map and reduce slots allocated to job J after the node failure. If the failed resources are not replenished, then the

number of map and reduce slots is correspondingly decreased. The number of map and reduce tasks yet to be processed are N^J_{M,t_f} and N^J_{R,t_f} as shown above. Then the performance bounds on the processing time of these tasks can be computed using Eq. 3.5 and Eq. 3.6. The worker failure is detected only after time δ depending on the value of the *heart beat interval*. Hence, the time bounds are also increased by the additional time delay δ.

3.7 Estimating Resources for a Given Deadline

In this section, we design an efficient procedure to estimate the minimum number of map and reduce slots that need to allocated to a job so that it completes within a given (soft) deadline.

3.7.1 SLO-based Performance Model

When users plan the execution of their MapReduce applications, they often have some *service level objectives* (SLOs) that the job should complete within time T. In order to support the job SLOs, we need to be able to answer a complementary performance question: given a MapReduce job J with input dataset D, what is the minimum number of map and reduce slots that need to be allocated to this job that it finishes within T?

There are a few design choices for answering this question:

- T is targeted as a *lower bound* of the job completion time. Typically, this leads to the least amount of resources that are allocated to the job for finishing within deadline T. The lower bound corresponds to "ideal" computation under allocated resources and is rarely achievable in real environments.

- T is targeted as an *upper bound* of the job completion time. This would lead to a more aggressive resource allocations and might result in a job completion time that is much smaller (better) than T because worst case scenarios are also rare in production settings.

- T is targeted as the *average* between lower and upper bounds on the job completion time. This solution might provide a balanced resource allocation that is closer for achieving the job completion time T.

Algorithm 1 finds the minimal combinations of map/reduce slots (S_M^J, S_R^J) for one of design choices above, e.g., when T is targeted as a *lower bound* of the job completion time. The algorithm sweeps through the entire range of map slot allocations and finds the corresponding values of reduce slots that are needed to complete the job within time T using a variation of the lower bound equation introduced in Section 3.3. The other cases when T is targeted as the upper bound and the average bound are handled similarly.

Algorithm 1 Resource Allocation Algorithm

Input:
Job profile of J
$(N_M^J, N_R^J) \leftarrow$ Number of map and reduce tasks of J
$(S_M, S_R) \leftarrow$ Total number of map and reduce slots in the cluster
$T \leftarrow$ Deadline by which job must be completed
Output: $P \leftarrow$ Set of plausible resource allocations (S_M^J, S_R^J)

for $S_M^J \leftarrow \text{MIN}(N_M^J, S_M)$ **to** 1 **do**
    Solve the equation $\frac{A_J^{low}}{S_M^J} + \frac{B_J^{low}}{S_R^J} = T - C_J^{low}$ for S_R^J
    if $0 < S_R^J \leq S_R$ **then**
        $P \leftarrow P \cup (S_M^J, S_R^J)$
    else
        // Job cannot be completed within deadline T
        // with the allocated map slots
        Break out of the loop
    end if
end for

The allocations of map and reduce slots to job J (with a known profile) for meeting soft deadline T are found using a variation of Eq. 4.1 introduced in Section 6.4, where A_J^{low}, B_J^{low}, and C_J^{low} are defined.

$$A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} = T - C_J^{low} \tag{3.12}$$

Let us use a simplified form of this equation shown below:

$$\frac{a}{m} + \frac{b}{r} = D \tag{3.13}$$

where m is the number of map slots, r is the number of reduce slots allocated to the job J, and a, b and D represent the corresponding constants (expressions) from Eq. 3.12. This equation yields a hyperbola if m and r are the variables. All integral points on this hyperbola are possible allocations

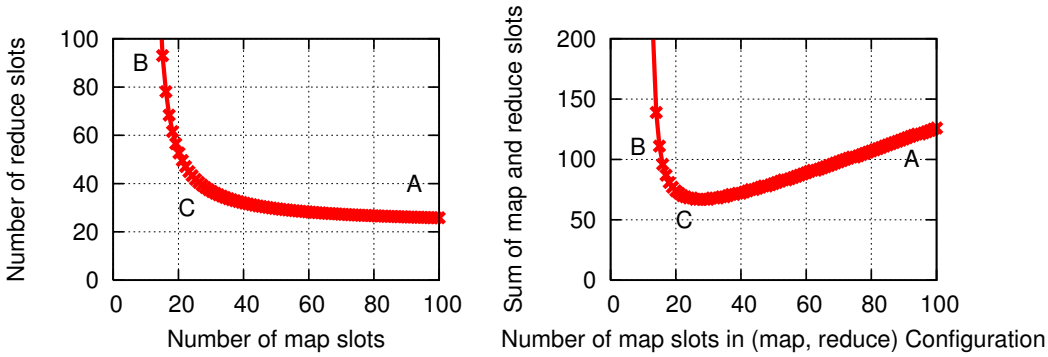


Figure 3.4: Lagrange curve

of map and reduce slots which result in meeting the same SLO deadline T. As shown in Figure 3.4 (left), the allocation could use the maximum number of map slots and very few reduce slots (shown as point A) or very few map slots and the maximum number of reduce slots (shown as point B). These different resource allocations lead to different amount of resources used (as a combined sum of allocated map and reduce slots) shown Figure 3.4 (right). There is a point where the sum of the map and reduce slots is minimized (shown as point C). We will show how to calculate this minima on the curve using Lagrange's multipliers, since we would like to conserve the map and reduce slots allocated to job J.

We wish to minimize $f(m, r) = m + r$ over $\frac{a}{m} + \frac{b}{r} = D$.
We set $\Lambda = m + r + \lambda\frac{a}{m} + \lambda\frac{b}{r} - D$.
Differentiating Λ partially with respect to m, r and λ and equating to zero, we get

$$\frac{\partial\Lambda}{\partial m} = 1 - \lambda\frac{a}{m^2} = 0 \tag{3.14}$$

$$\frac{\partial\Lambda}{\partial r} = 1 - \lambda\frac{b}{r^2} = 0 \tag{3.15}$$

$$\frac{\partial\Lambda}{\partial\lambda} = \frac{a}{m} + \frac{b}{r} - D = 0 \tag{3.16}$$

Solving these equations simultaneously, we get

$$m = \frac{\sqrt{a}(\sqrt{a} + \sqrt{b})}{D}, \quad r = \frac{\sqrt{b}(\sqrt{a} + \sqrt{b})}{D} \tag{3.17}$$

These values are the optimal values of map and reduce slots such that the

number of slots used is minimized while meeting the deadline. In practice, these values have to be integral. Hence, we round up the values found by these equations and use them as an approximation.

SLO (s)	Allocated (map, reduce) slots based on		
	Lower bound	Average bound	Upper bound
240	(41, 21)	(58, 30)	(64, 64)
300	(33, 17)	(43, 22)	(63, 32)
360	(27, 14)	(34, 18)	(45, 23)
420	(24, 12)	(28, 15)	(35, 18)

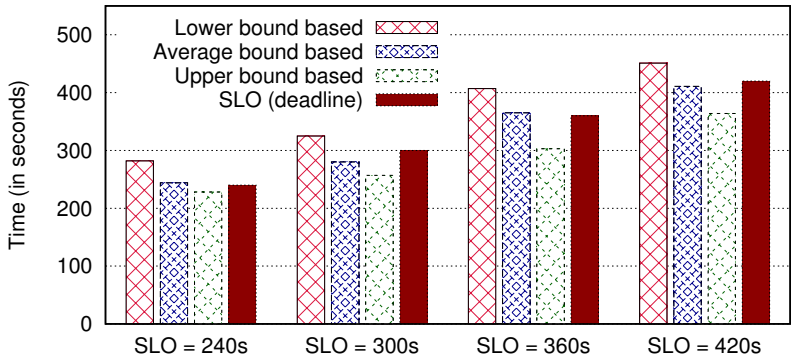


Figure 3.5: Slot allocations and job completion times based on minimum resource allocation based on different bounds.

3.7.2 Initial Evaluation of SLO-based Model

In this section, we perform an initial set of performance experiments to validate the SLO-based model introduced in Section 3.7.1. We aim to evaluate the accuracy of resource allocations recommended by the model for completing the job within a given deadline T. For validation we use the *WikiTrends* application (see Section 4.3.2 for more details).

The WikiTrends application consists of 71 map and 64 reduce tasks. We configure one map and reduce slot on each machine. We vary the SLO (deadline) for the job through 4, 5, 6 and 7 minutes. Using the lower, average and upper bound as the target SLO, we compute the minimum number of map and reduce slot allocations as shown in the table in Figure 3.5. Using these map and reduce slot allocations, we execute the WikiTrends application and measure the job completion times as shown in Figure 3.5. The model based on the lower bound suggests insufficient resource allocations: the job

executions with these allocations missed their deadlines. The model based on the upper bound aims to over provision resources (since it aims to "match" the worst case scenario). While all the job executions meet the deadlines, the measured job completion times are quite lower than the target SLO. We observe that the average bound based allocations result in job completion times which are closest to the given deadlines: within 7% of the SLO.

3.8 ARIA Implementation

Our goal is to propose a novel SLO-scheduler for MapReduce environments that supports a new API: a job can be submitted with a desirable job completion deadline. The scheduler will then estimate and allocate the appropriate number of map and reduce slots to the job so that it meets the required deadline. To accomplish this goal we designed and implemented a framework, called *ARIA*, to address this problem. The implementation consists of the following five interacting components shown in Figure 3.6:

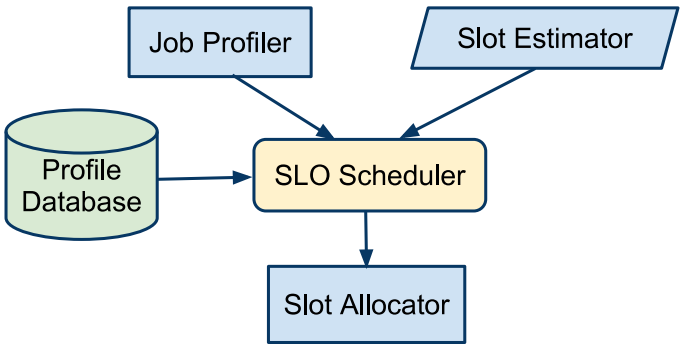


Figure 3.6: ARIA implementation.

1. **Job Profiler:** It collects the job profile information for the currently running or finished jobs. We use the Hadoop counters which are sent from the workers to the master along with each heartbeat to build the profile. This profile information can also be gleaned from the logs in the HDFS output directory or on the job master after the job is completed. The job profile is then stored persistently in the profile database.

2. **Profile Database:** We use a MySQL database to store the past

profiles of the jobs. The profiles are identified by the (user, job name) which can be specified by the application.

3. **Slot Estimator:** Given the past profile of the job and the deadline, the slot estimator calculates the minimum number of map and reduce slots that need to be allocated to the job in order to meet its SLO. Essentially, it uses the Lagrange's method to find the minima on the allocation curve introduced in Section 3.7.1.

4. **Slot Allocator:** Using the slots calculated from the slot estimator, the slot allocator assigns tasks to jobs such that the job is always below the allocated thresholds by keeping track of the number of running map and reduce tasks. In case there are spare slots, they can be allocated based on the additional policy. There could be different classes of jobs: jobs with/without deadlines. We envision that jobs with deadlines will have higher priorities for cluster resources than jobs without deadlines. However, once jobs with deadlines are allocated their required minimums for meeting the SLOs, the remaining slots can be distributed to the other job classes.

5. **SLO-Scheduler:** This is the central component that co-ordinates events between all the other components. Hadoop provides support for a pluggable scheduler. The scheduler makes global decisions of ordering the jobs and allocating the slots across the jobs. The scheduler listens for events like job submissions, worker heartbeats, etc. When a heartbeat containing the number of free slots is received from the workers, the scheduler returns a list of tasks to be assigned to it.

The SLO-scheduler has to answer two inter-related questions: which job should the slots be allocated and how many slots should be allocated to the job? The scheduler executes the *Earliest Deadline First* algorithm (EDF) for job ordering to maximize the utility function of all the users. The second (more challenging) question is answered using the Lagrange computation discussed in Section 3.7. The detailed slot allocation schema is shown in Algorithm 2.

As shown in Algorithm 2, it consists of two parts: 1) when a job is added, and 2) when a heartbeat is received from a worker. Whenever a job is added, we fetch its profile from the database and compute the minimum

Algorithm 2 Earliest Deadline First Algorithm

1: **When job** j **is added:**
2: Fetch $Profile_j$ from database
3: Compute minimum number of map and reduce slots (m_j, r_j) using Lagrange's multiplier method
4: **When a heartbeat is received from node** n**:**
5: Sort $jobs$ in order of earliest deadline
6: **for each** slot s in free map/reduce slots on node n **do**
7: **for each** job j in $jobs$ **do**
8: **if** $RunningMaps_j < m_j$ **and** s is map slot **then**
9: **if** job j has unlaunched map task t with data on node n **then**
10: Launch map task t with local data on node n
11: **else if** j has unlaunched map task t **then**
12: Launch map task t on node n
13: **end if**
14: **end if**
15: **if** $FinishedMaps_j > 0$ **and** s is reduce slot **and** $RunningReduces_j < r_j$ **then**
16: **if** job j has unlaunched reduce task t **then**
17: Launch reduce task t on node n
18: **end if**
19: **end if**
20: **end for**
21: **end for**
22: **for each** task T_j finished slots by node n **do**
23: Recompute (m_j, r_j) based on the current time, current progress and deadline of job j
24: **end for**

number of map and reduce slots required to complete the job within its specified deadline using the Lagrange's multiplier method discussed earlier in Section 3.7.1.

Workers periodically send a heartbeat to the master reporting their health, the progress of their running tasks and the number of free map and reduce slots. In response, the master returns a list of tasks to be assigned to the worker. The master tracks the number of running and finished map and reduce tasks for each job. For each free slot and each job, if the number of running maps is lesser than the number of map slots we want to assign it, a new task is launched. As shown in Lines 9 - 13, preference is given to tasks that have data local to the worker node. Finally, if at least one map has finished, reduce tasks are launched as required.

In some cases, the amount of slots available for allocation is less than required minima for job J and then J is allocated only a fraction of required resources. As time progresses, the resource allocations are recomputed during the job's execution and adjusted if necessary as shown in Lines 22-24 (this is a very powerful feature of the scheduler that can increase resource allocation if the job execution progress is behind the targeted and expected one). Whenever a worker reports a completed task, we decrement N_M^J or N_R^J in the SLO-based model and recompute the minimum number of slots.

3.9 ARIA Evaluation

In this section, we evaluate the efficiency of the new SLO-scheduler using a set of realistic workloads. First, we motivate our evaluation approach by a detailed analysis of the simulation results. Then we validate the simulation results by performing similar experiments in the 66-node Hadoop cluster.

3.9.1 Workload

Our experimental workload consists of a set of representative applications that are run concurrently. We can run the same application with different input datasets of varying sizes. A particular application reading from a particular set of inputs is called an application instance. Each application instance can be allocated varying number of map and reduce slots resulting

in different job executions. The applications used in our experiments are as follows:

1. **Word count:** This application computes the occurrence frequency of each word in the Wikipedia article history dataset. We use three datasets of sizes: 32GB, 40GB and 43GB.

2. **Sort:** This applications sorts a set of records that is randomly generated. The application uses identity map and identity reduce functions as the MapReduce framework does the sorting. We consider three instances of Sort: 8GB, 64GB and 96GB.

3. **Bayesian classification:** We use a step from the example of Bayesian classification trainer in Mahout[7]. The mapper that extracts features from the input corpus and outputs the labels along with a normalized count of the labels. The reduce performs a simple addition of the counts and is also used as the combiner. The input dataset is the same Wikipedia article history dataset, except the chunks split at page boundaries.

4. **TF-IDF:** The Term Frequency - Inverse Document Frequency application is often used in information retrieval and text mining. It is a statistical measure to evaluate how important a word is to a document. We used the TF-IDF example from Mahout and used the same Wikipedia articles history dataset.

5. **WikiTrends:** This application is described in detail in Section 4.1, since it is used in the initial evaluation.

6. **Twitter:** This application uses the 25GB twitter dataset created by [23] containing an edge-list of twitter userids. Each edge (i, j) means that user i follows user j. The Twitter application counts the number of asymmetric links in the dataset, that is, $(i, j) \in E$, but $(j, i) \notin E$. We use three instance processing 15GB, 20GB and 25GB respectively.

[7] http://http://mahout.apache.org/

3.9.2 Performance Invariants

In our first set of experiments, we aim to validate whether the metrics, that we chose for the inclusion in the job profile, indeed represent performance invariants across different executions of the job on the same input dataset. To this end, we execute our MapReduce jobs on the same datasets and the same Hadoop cluster but with a variable number of map and reduce slots: *i)* 64 map and 32 reduce slots, *ii)* 16 map and 16 reduce slots. We observe that the average duration metrics are within 5% of each other. The maximum durations show slightly higher variance. Each experiment is performed 10 times, and again, collected metrics exhibit less than 5% variation. From these measurements, we conclude that job profile indeed accurately captures application behavior characteristics and reflect the job performance invariants.

3.9.3 Scaling Factors

We execute WikiTrends and WordCount applications on gradually increasing datasets with a fixed number of reduce tasks for each application. Our intent is to measure the trend of the shuffle and reduce phase durations (average and maximum) and validate the linear regression approach proposed in Section 3.5. The following table gives the details of the experiments and the resulting co-efficients of linear regression, i.e., scaling factors of shuffle and reduce phase durations derived for these applications.

Parameters	**WikiTrends**	**WordCount**
Size of input dataset	4.3GB to 70GB	4.3GB to 43GB
Number of map tasks	70 to 1120	70 to 700
Number of reduce tasks	64	64
Number of map, reduce slots	64, 32	64, 32
$C^{Sh}_{0,avg}, C^{Sh}_{1,avg}$	16.08, 2.44	6.92, 0.66
$C^{Sh}_{0,max}, C^{Sh}_{1,max}$	10.75, 2.29	11.28, 0.71
$C^{R}_{0,avg}, C^{R}_{1,avg}$	11.45, 0.56	4.09, 0.22
$C^{R}_{0,max}, C^{R}_{1,max}$	7.96, 0.43	7.26, 0.24

Figure 3.7 shows that the trends are indeed linear for WikiTrends and WordCount. Note that the lines do not pass through the origin and hence the durations are not directly proportional to the dataset size.

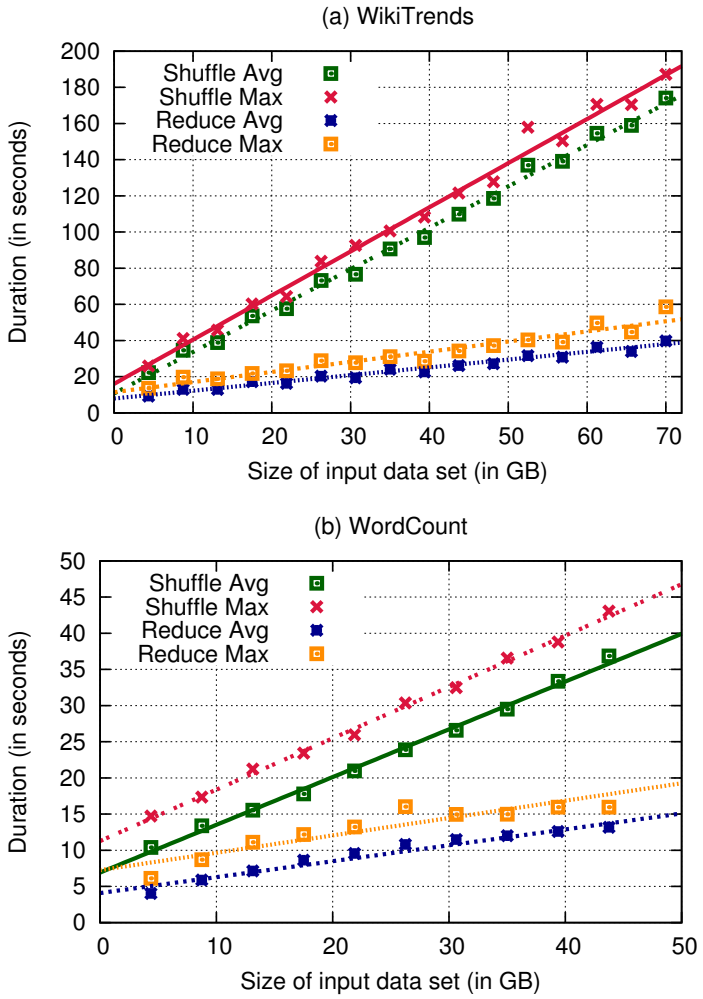


Figure 3.7: Linear scaling of shuffle and reduce durations for WikiTrends and WordCount.

We observe similar results for Grep and Sort applications but do not include them in the paper due to lack of space.

3.9.4 Performance Bounds of Job Completion Times

In section 6.4, we designed performance bounds that can be used for estimating the completion time of MapReduce application with a given job profile. The expectations are that the job profile can be built using a set of job executions for processing small size input datasets, and then this job profile can be used for predicting the completion time of the same application pro-

cessing a larger input dataset. Therefore, in these experiments, first, the job profiles are built using the three trials on small datasets (e.g., 4.3, 8.7 and 13.1 GB for WordCount) with different numbers of map and reduce slots. After that, by applying linear regression to the extracted job profiles from these runs, we determine the scaling factors for shuffle and reduce phases of our MapReduce jobs. The derived scaling factors are used to represent the job performance characteristics and to extrapolate the duration of the shuffle and reduce phases when the same applications are used for processing larger input datasets with parameters shown in the following table:

Parameters	**Twitter**	**Sort**	**WikiTrends**	**WordCount**
# of map tasks	370	1024	168	425
# of reduce tasks	64	64	64	64
# of map slots	64	64	64	64
# of reduce slots	16	32	8	8

Finally, by using the updated job profiles and applying the formulae described in Section 6.4, we predict the job completion times.

The results of these experiments are shown in Figure 3.8. We observe that the relative error between the predicted average T_J^{avg} and the measured job completion time is less than **10%** in all cases. However, for three applications (Sort, WikiTrends, and WordCount) the average time is below the measured one. We believe that the set of experiments in the staging environment should help to choose which bound (or weighted combination of them) should be used for more accurate estimate of the job completion time. The predicted upper bound on the job completion time T_J^{up} can be used for ensuring SLOs. The solid fill color within the bars in

Figure 3.8 represent the reduce stage duration, while the pattern portion reflects the duration of the map stage. For Grep, Sort, and WordCount, bounds derived from the profile provide a good estimate for map and reduce stage durations. For WikiTrends, we observe a higher error in the estimation of the durations, mostly, due to the difference in processing of the unequal compressed files as inputs.

The power of the proposed approach is that it offers a compact job profile that can be derived in a small staging environment and then used for completion time prediction of the job on a large input dataset while also using different amount of resources assigned to the job.

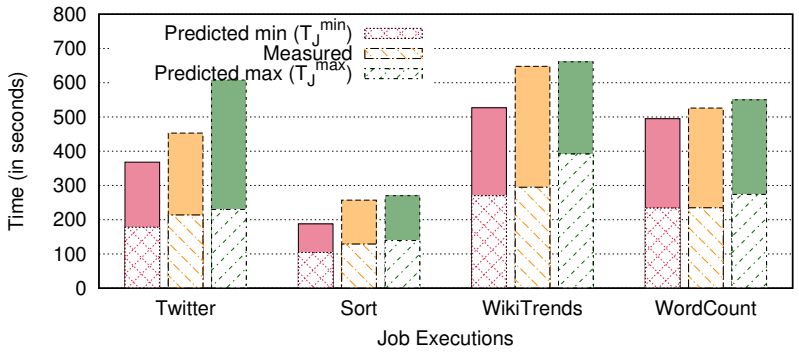


Figure 3.8: Comparison of predicted and measured job completion times.

3.9.5 SLO-based Resource Provisioning

In this section, we perform experiments to validate the accuracy of the SLO-based resource provisioning model introduced in Section 3.7. It operates over the following inputs *i)* a job profile built in the staging environment using smaller datasets, *ii)* the targeted amount of input data for processing, *iii)* the required job completion time. We aim to evaluate the accuracy of resource allocations recommended by the model for completing the job within a given deadline.

Figure 3.9 shows a variety of plausible solutions (the outcome of the SLO-based model) for Grep, WikiTrends and WordCount with a given deadline D= 5, 9, and 8 minutes respectively. The X and Y axes of the graph show the number of map and reduce slots respectively that need to be allocated in order to meet the job's deadline. Figure 3.9 presents three curves that correspond to three possible design choices for computing the required map/reduce slots as discussed in Section 3.7: when the given time T is targeted as the lower bound, upper bound, or the average of the lower and upper bounds. As expected, the recommendation based on the upper bound (worst case scenario) suggests more aggressive resource allocations with a higher number of map and reduce slots as compared to the resource allocation based on the lower bound. The difference in resource allocation is influenced by the difference between the lower and upper bounds. For example, Grep has very tight bounds which lead to more similar resource allocations based on them. For WikiTrends the difference between the lower and upper bounds of completion time estimates is wider, which leads to a larger difference in the

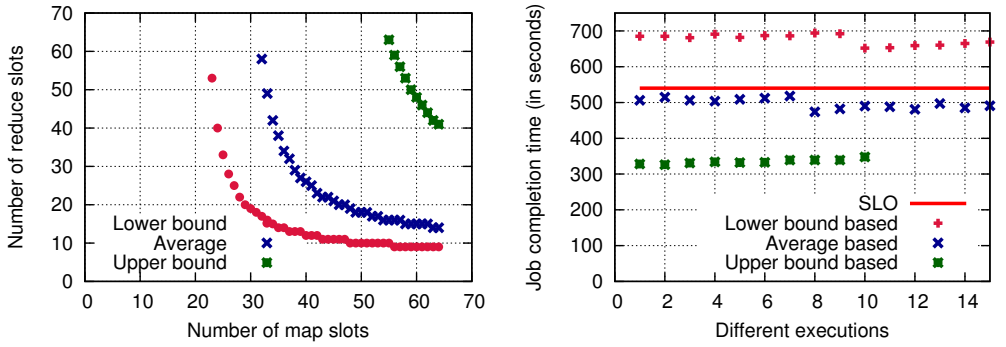
resource allocation options.

Next, we perform a set of experiments with the applications on our 66-node Hadoop cluster. We sample each curve in Figure 3.9, and execute the applications with recommended allocations of map and reduce slots in our Hadoop cluster to measure the actual job completion times. Figure 3.9 summarizes the results of these experiments. If we base our resource computation on the lower bound of completion time, it corresponds to the "ideal" scenario. The model based on lower bounds suggests insufficient resource allocations: almost all the job executions with these allocations have missed their deadline. The closest results are obtained if we use the model that is based on the average of lower and upper bounds of completion time. However, in many cases, the measured completion time can exceed a given deadline (by 2-7%). If we base our computation on the upper bounds of completion time, the model over provisions resources. While all the job executions meet their deadline, the measured job completion times are lower than the target SLO, often by as much as 20%. The resource allocation choice will depend on the user goals and his requirements on how close to a given SLO the job completion time should be. The user considerations might also take into account the service provider charging schema to evaluate the resource allocation alternatives on the curves shown in Figure 3.9.

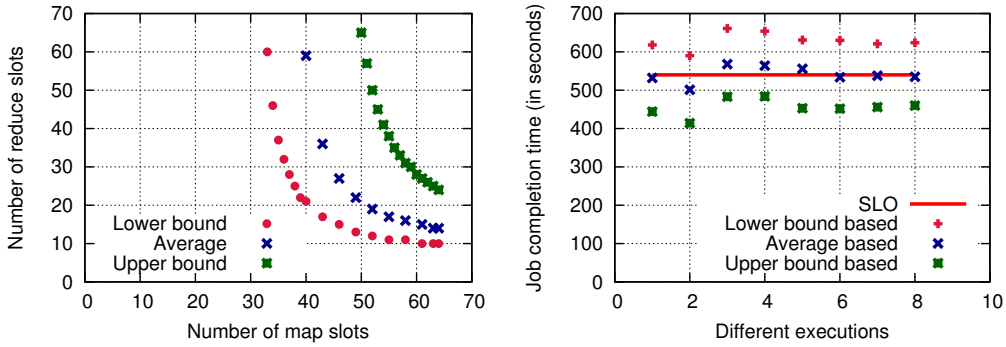
3.9.6 Prediction of Job Completion Time with Failures

In this section, we validate the model for predicting the job completion time with failures introduced in Section 3.6. For this experiment, we set the heartbeat interval to $3s$. If a heartbeat is not received in the last $20s$, the worker node is assumed to have failed. We use the WikiTrends application which consists of 720 map and 120 reduce tasks. The application is allocated 60 map and 60 reduce slots. The WikiTrends execution with given resources takes $t = 1405s$ to complete under normal circumstances. Figure 3.10 shows a set of two horizontal lines that correspond to lower and upper bounds of the job completion time under normal case.

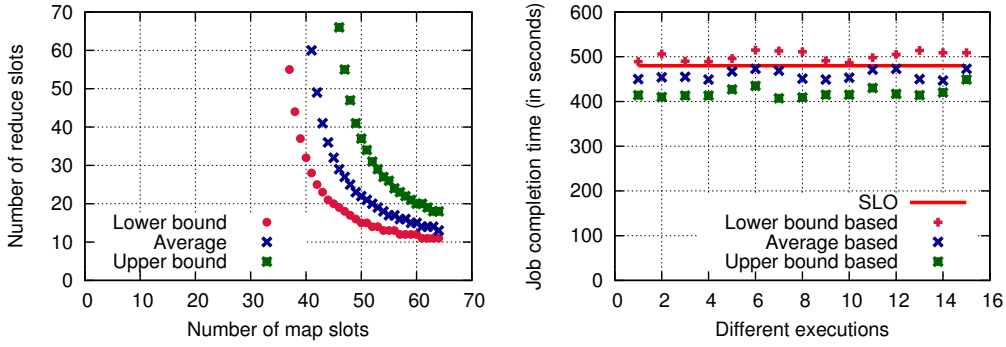
Then, using the model with failures introduced in Section 3.6, we compute the lower and upper bounds for job completion time when a failure happens at time t_f (time is represented by X-axes). The model considers two different



(a) Twitter (Deadline = 9 mins)



(b) WikiTrends (Deadline = 9 mins)



(c) WordCount (Deadline = 8 mins)

Figure 3.9: Allocation curves and completion times based on bounds for different deadlines across three applications.

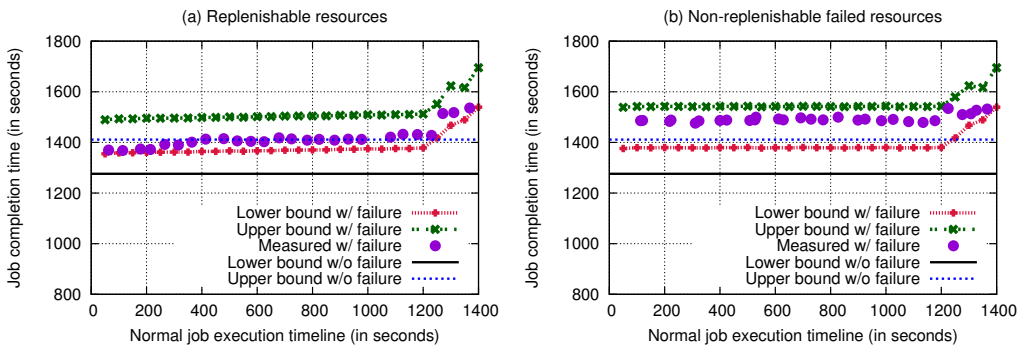


Figure 3.10: Model with failures: two cases with replenishable resources and non-replenishable failed resources.

scenarios: when resources of the failed node are 1) replenished and 2) not replenished. These scenarios are reflected in Figure 3.10 (a) and (b) respectively. Figure 3.10 shows the predicted lower and upper bounds (using the model with failures) along with the measured job completion time when the worker process is killed at different points in the course of the job execution.

The shape of the lines (lower and upper bounds) for job completion time with failures is quite interesting. While the completion time with failures increases compared to the regular case, but this increase is practically constant until approximately $t = 1200s$. The map stage completes at $t = 1220(\pm 10)s$. So, the node failure during the map stage has a relatively mild impact on the overall completion time, especially when the failed resources are replenished. However, if the failure happens in the reduce stage (especially towards the end of the job processing) then it has a more significant impact on the job completion time even if the failed node resources are replenished. Note that the measured job completion time with failures stays within predicted bounds, and hence the designed model can help the user to estimate the worst case scenario.

3.9.7 Simulation

We implement a discrete event simulator in order to understand the efficacy of our scheduling and resource allocation algorithm. We do not simulate details of worker nodes (their hard disks or network packet transfers) as it is done

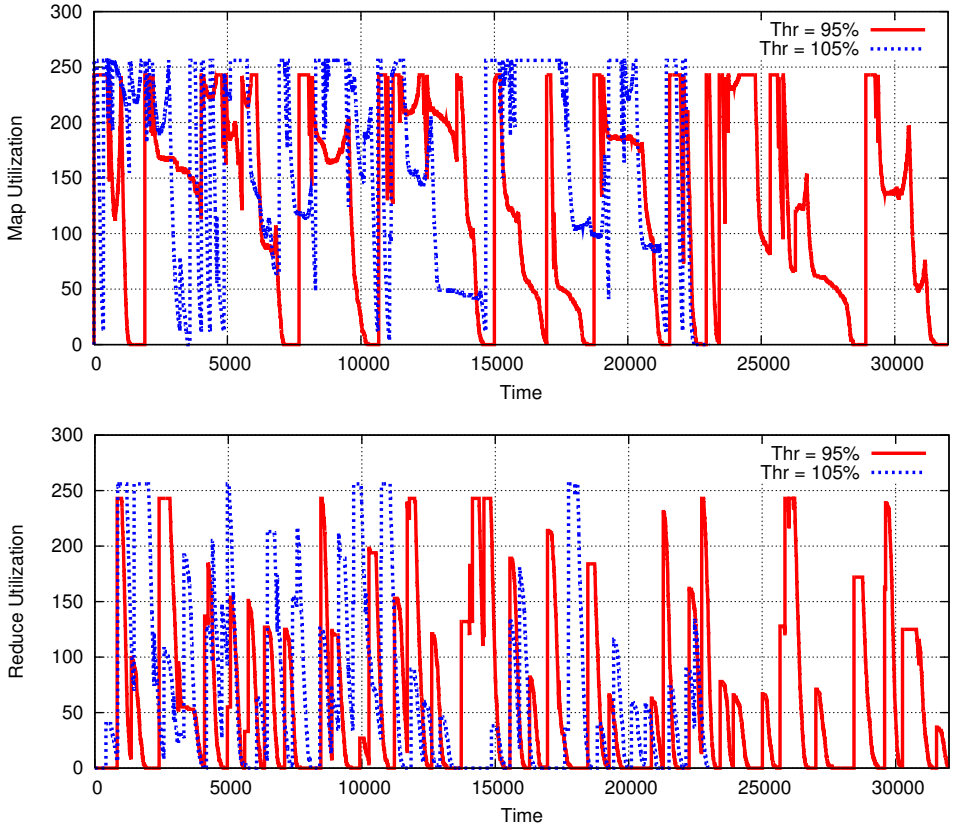


Figure 3.11: Cluster load over time in the simulations with *Yahoo!* workload: a) map slots, b) reduce slots.

in MRPerf [24], because we use job profiles to represent job latencies during different phases of MapReduce processing in the cluster. We concentrate on simulating the job master decisions and the task/slot allocations across multiple jobs.

We maintain data structures similar to the Hadoop job master such as the *job queue*: a priority queue of jobs sorted by the earliest deadline first. Since the slot allocation algorithm makes a new decision when a map or reduce task completes, we simulate the jobs at the task level. The simulator maintains priority queues for *seven event types*: job arrivals and departures, map and reduce task arrivals and departures, and a timer event (used for accounting purposes).

In the simulations (and later in the testbed evaluations), we will assess the quality of scheduling and resource allocations decisions by observing the following metric. Let the execution consist of a given set of n jobs $J_1, J_2, \ldots, J_n$ with corresponding deadlines $D_1, D_2, \ldots, D_n$, and known job profiles. Let these jobs be completed at times $T_1, T_2, \ldots, T_n$, and let Θ be the set of all jobs whose deadline has been exceeded. Then we compute the following utility function:

$$\sum_{J \in \Theta} \frac{T_J - D_J}{D_J}$$

This function denotes the the sum of the *relative deadlines exceeded*. We are interested in minimizing this value.

We perform simulations with two different workloads denoted as W_1 and W_2.

Workload W_1 represents a mix of the six realistic applications with different input dataset sizes as introduced and described in Section 4.3.3. We refer to W_1 as the *testbed workload*. It aims to simulate the workload that we use for SLO-scheduler evaluation in our 66-node Hadoop testbed.

Workload W_2 (called *Yahoo!* workload) represents a mix of MapReduce jobs that is based on the analysis of the M45 Yahoo! cluster [25], and that is generated as follows:

- The job consists of the number of map and reduce tasks defined by the distributions $N(154, 558)$ and $N(19, 145)$ respectively, where $N(\mu, \sigma)$ is the *normal distribution* with mean μ and standard deviation σ.

- Map task durations are defined by $N(100, 20)$ and reduce task durations

are from $N(300, 30)$, because reduce tasks are usually longer than map tasks since they perform shuffle, sort and reduce. (The study in [25] did not report the statistics for individual map and reduce task durations, so we chose them as described above).

- The job deadline (which is relative to the job completion time) is set to be uniformly distributed in the interval $[T_J, 3 \cdot T_J]$, where T_J is the completion time of job J given all the cluster resources.

The job deadlines in W_1 and W_2 are generated similarly. Table 3.2 provides the summary of W_1 and W_2.

Set	**App**	**Number of map tasks**	**Map task duration**	**Reduce task duration**
W1	Bayes	54, 68, 72	436s	33s
	Sort	256, 512, 1024	9s	53s
	TF-IDF	768	11s	66s
	Twitter	294, 192, 390	59s	65s
	Wikitrends	71, 720, 740	179s	79s
	WordCount	507, 640, 676	56s	21s
W2	Yahoo!	$N(154, 558)$	$N(100, 20)$	$N(300, 30)$

Table 3.2: Simulation workloads W_1 and W_2.

To understand the effectiveness of resource allocation by the SLO-scheduler and the quality of its decisions we aim to create varying load conditions. However, instead of spawning jobs with an inter-arrival distribution, we spawn a new job so as to keep the load in the cluster below a certain threshold. Since our goal is to evaluate the efficiency of the SLO-scheduler decisions, we would like to generate the arrival of jobs that have realistic chances of completing in time. If the cluster does not have enough resources for processing of the newly arrived job then the scheduler might fail to allocate sufficient resources, and the job might not be able to meet its deadline. However, in this case, it is not the scheduler's fault. Moreover, as we will demonstrate later, there is a drastic difference between the peak and average load measured in the simulated cluster over time, which provides an additional motivation to design a job arrival process driven by the load threshold. We define the load as the sum of the percentage of running map and reduce tasks compared to the total cluster capacity. We spawn a new job whenever its minimum pair

computed by the Lagrange method combined with the current cluster load is below the threshold.

The simulator was very helpful in designing the ARIA evaluation approach. We were able to generate, execute, and analyze many different job arrival scenarios and their outcomes before deciding on the job arrivals driven by a threshold. Simulations take a few minutes compared to multi-hour executions of similar workloads in the real testbed.

Table 3.3 summarizes the results of our simulations for *Yahoo!* and *testbed* workloads with 100 jobs. Each experiment was repeated 100 times. In the table, we report the simulation results averaged across these 100 runs.

Load threshold for arrival (%)	**SLO exceeded utility (%)**		**# of jobs with missed SLO**		**Average Load (%)**	
Workload	W1	W2	W1	W2	W1	W2
105	3.31	12.81	0.54	5.21	28.15	34.31
100	1.41	4.65	0.43	3.54	26.51	33.30
95	0	0	0	0	25.21	30.77
90	0	0	0	0	24.70	29.43
85	0	0	0	0	23.52	28.47

Table 3.3: Simulation results with *testbed* ($W1$) and *Yahoo!* ($W2$) workloads.

We observe that allocation decisions of the SLO-scheduler enables the job to efficiently meet the job deadlines when the load threshold is less than 100%. Moreover, even with a higher load threshold (105%) we see that only a few jobs are missing their deadlines. The last column reports the average utilization of the cluster during the runs measured as the percentage of running map and reduce tasks compared to the total cluster capacity. It is significantly lower than the load threshold used for job arrivals. Figure 3.11 shows the number of running map and reduce tasks in the cluster over time when processing the *Yahoo!* workload for two different simulations: with load threshold of 95% and 105%. It shows that the average load can be a misleading metric to observe: the individual map and reduce slots' utilization might be quite high, but since the reduce tasks do not start till the map tasks are completed for a given job this can lead to a low average utilization in the cluster. A similar situation is observed for simulations with *testbed* workload (W_1). We omit the figure due to lack of space.

We observe the similarity of simulation results for two quite different workload sets, that makes us to believe in the generality of presented conclusions.

3.9.8 Testbed Evaluation of ARIA

For evaluating ARIA and validating the efficiency of resource allocation decisions of the new SLO-scheduler in our 66-node Hadoop cluster, we used applications described in Section 4.3.3 that constitute the *testbed workload* W_1 as summarized in Table 3.2.

First, we executed each of the applications in isolation with their different datasets and three different map and reduce slot allocations. This set of experiments was run three times and the job profiles and the variation in the averages was less than 10% (up to 20% variation was seen in the maxima and minima). These experiments and the results are similar to ones we performed in our initial performance evaluation study presented in Section 3.4. Then, we also executed these applications along with each other, and the extracted job profiles show a slightly higher variation while still being very close to the earlier extracted job profiles.

Using the same evaluation approach and technique designed during our simulation experiments and described in detail in Section 3.9.7, we maintain the load on the testbed cluster below a certain threshold for generating varying load conditions. To assess the quality of scheduling and resource allocation decisions, we observe the number of jobs exceeding deadlines and measure the relative deadlines exceeded. The results are summarized in Table 3.4.

Load threshold for arrival (%)	**SLO exceeded utility (%)**	**# of jobs with missed SLOs**	**Average Load (%)**
105	7.62	1	29.27
100	4.58	1	27.34
95	0	0	26.46
90	0	0	25.63
85	0	0	24.39

Table 3.4: Results of *testbed workload* execution.

We observe that a very few jobs miss their deadlines for load threshold

above 100% with relatively low numbers for exceeded deadlines. We also measured the accuracy of job completions with respect to the given jobs deadlines and observe that they range in between 5%-15%, which is slightly worse than the simulation results, but close to our initial evaluations presented in Figure 3.5. Average utilization measured in the testbed is very close to simulation results, that further validates the accuracy of our simulator.

The performance overhead of the scheduling in ARIA is negligible: it takes less than 1 second for scheduling 500 jobs on our 66 node cluster. Likewise, the logging/accounting infrastructure is enabled by default on production clusters and can be used for generating job profiles.

We do not compare our new scheduler with any other existing schedulers for Hadoop or proposed in literature, because all these schedulers have very different objective functions for making scheduling decisions. For example, it would not be useful to compare our scheduler with the Hadoop Fair Scheduler (HFS) [20] because HFS aims to support a fair resource allocation across the running jobs and does not provide the targeted resource allocations for meeting the jobs SLOs. As a result, the HFS scheduler might have a high number of jobs with missed deadlines and arbitrarily high SLO-exceeded utility function.

3.10 Summary

In the enterprise setting, sharing a MapReduce cluster among multiple applications is a common practice. Many of these applications need to achieve performance goals and SLOs, that are formulated as the completion time guarantees. In this work, we propose a novel framework *ARIA* to address this problem. It is based on the observation that we can profile a job that runs routinely and then use its profile in the designed MapReduce performance model to estimate the amount of resources required for meeting the deadline. These resource requirements are enforced by the SLO-scheduler.

Chapter 4

SimMR: Simulation Framework for MapReduce

4.1 Motivation

One of the challenging tasks in shared MapReduce environments is the ability to tailor and efficiently control resource allocations among different jobs for achieving their performance goals. Often, the jobs are partitioned in different classes of service (e.g., platinum, silver, and bronze at Facebook [26]) and then processed by different Hadoop clusters with specially created management and resource allocation strategies. This is mainly done in order to guarantee performance isolation and have a predictable completion time for production jobs. However, when there is a need to expand the set of production jobs with new applications and additional data processing, first, one has to evaluate whether additional resources are required, and then how they should be allocated for meeting performance goals of the jobs in the extended set. To assist system administrators in performance evaluation and simplify MapReduce cluster management, new fast and accurate tools are needed.

In the past couple of years, job scheduling and workload management issues in MapReduce environments have received much attention. Currently, there are at least three different schedulers broadly used for job processing: the default FIFO scheduler, the *Capacity* scheduler [27], and the *Hadoop Fair Scheduler* (HFS) [20]. Each scheduler's decisions are based on several factors like simplicity, throughput, fairness, data locality, capacity guarantee, etc. Moreover, there are several research prototypes, e.g., FLEX [28], Dynamic Priority (DP) scheduler [29], ARIA [30], that aim to enhance the existing schedulers by exploiting new principles and performance models for supporting additional features.

Designing, prototyping, and evaluating new resource allocation and job scheduling algorithms in large-scale distributed systems such as Hadoop is

a challenging, labor-intensive, and time-consuming task. Experiments performed in a real MapReduce testbed can take hours (to days) to obtain any preliminary results. Such evaluation is often limited to a set of specific applications (or benchmarks) available for experimentation. These experiments cannot be performed in production clusters of interest. Our goal is to design an accurate and fast simulation environment for evaluating workload management and resource optimization decisions in MapReduce environments. It will assist Hadoop cluster administrators in their daily tasks, helping them avoid error-prone, guess-based decisions.

In this Chapter, we present a new MapReduce simulator, called *SimMR* (pronounced as *simmer*), that can replay execution traces of real workloads collected in Hadoop clusters (as well as synthetic traces based on statistical properties of workloads) for evaluating different resource allocation and scheduling ideas in MapReduce environments.

SimMR consists of the following three components:

1. **Trace Generator** – a module that generates a replayable workload trace by processing the job tracker logs or using a synthetic workload description.

2. **Simulator Engine** – a discrete event simulator that accurately emulates the Hadoop job master decisions for map/reduce slot allocations across multiple jobs.

3. **Scheduling policy** – a pluggable scheduling module that dictates the ordering of jobs and the amount of allocated resources to different jobs over time.

We validate the accuracy of SimMR by, first, executing a set of realistic MapReduce applications in a 66-node Hadoop cluster and then replaying the collected job execution traces in SimMR. The simulator accurately reproduces the original job processing with less than 2.7% average (6.6% maximum) error across the applications in the simulated set. We compare SimMR with the available open source, Apache's MapReduce simulator, called Mumak [31]. This simulator replays traces collected with a log processing tool, called Rumen [32]. In our evaluation study, we observe that Mumak's trace replay deviates significantly from the original job processing: Mumak's sim-

ulation exhibits 37% average (51.7% maximum) error while replaying the same traces.

The main difference between Mumak and SimMR is that Mumak omits modeling the shuffle/sort phase. For many applications this could lead to a significant error in completion time estimates and inaccurate workload modeling over time. We believe that the modeling framework proposed in SimMR for replaying the shuffle/sort and reduce phases could be adopted by Mumak to make it more accurate.

To assess the simulator speed, we collected traces from 1148 jobs run on our 66 node cluster during the last 6 months. The results show that SimMR replays these jobs in 1.5 seconds, while Mumak's execution takes 680 seconds to replay the same set of jobs. Thus, SimMR is two orders of magnitude faster than Mumak.

Finally, we present a case study with SimMR that is used for a fast (but accurate) performance analysis and comparison of two different deadline-driven Hadoop schedulers over a diverse set of workloads.

This chapter is organized as follows. Section 4.2 describes the design choices and overall architecture of SimMR. We evaluate SimMR in Section 4.3. Section 4.4 provides a case study with SimMR by comparing two different deadline schedulers. Section 4.5 discusses complementary mechanisms for deadline based workload management. Section 4.6 discusses three pieces of the workload management puzzle and how we put them together. Finally, we summarize our findings in Section 4.7.

4.2 SimMR Design

Our goal is to build a simulator which is capable of replaying the scheduling decisions over a large workload (several months of job logs) in a few minutes on a single machine. We focus on simulating the job master decisions and the task/slot allocations across multiple concurrent jobs. This would aid in understanding the efficacy of our scheduling and resource allocation algorithms. It is a non-goal to simulate details of the TaskTracker nodes (their hard disks or network packet transfers) as done by MRPerf [24]. Instead, we use job profiles (with task durations) to represent the latencies during different phases of MapReduce processing in the cluster.

Figure 4.1 shows the overall design of SimMR. The job traces can be generated using two methods. Firstly, they can be obtained from actual jobs executed on the real cluster using the **MRProfiler**. Alternatively, the trace can be synthetically generated using **Synthetic TraceGen** by observing the statistical properties of the workloads. These collected traces are stored persistently in the **Trace Database**. Using the job trace and a **Scheduling policy** as input, the **Simulator Engine** replays the trace by enforcing the scheduling and resource allocation decisions and generates the output log. Various scheduling policies, such as **FIFO**, **MinEDF** and **MaxEDF** that are considered in these paper, can be enforced by SimMR.

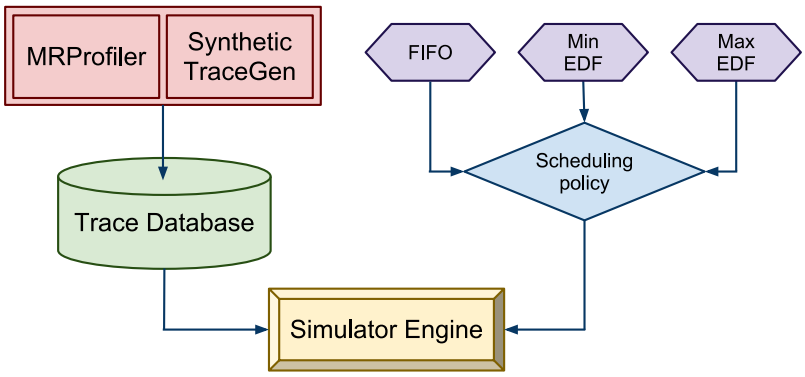


Figure 4.1: SimMR Design

4.2.1 Trace Generation

We can generate job traces using two methods: MRProfiler and Synthetic TraceGen. MRProfiler extracts the job performance metrics by processing the counters and logs stored at the JobTracker at the end of each job. The job tracker logs reflect the MapReduce jobs' processing in the Hadoop cluster. They faithfully record the detailed information about the map and reduce tasks' processing. The logs also have useful information about the shuffle/sort stage of each job. Alternatively, we can model the distributions of the durations based on the statistical properties of the workloads and generate synthetic traces using Synthetic TraceGen. This can help evaluate hypothetical workloads and consider what-if scenarios. We store job traces persistently in a Trace database (for efficient lookup and storage) using a *job template*. The job template summarizes the job's essential performance

characteristics during its execution in the cluster. We extract the following metrics for each job J:

- $(\mathbf{N_M^J}, \mathbf{N_R^J})$ - the number of map tasks and reduce tasks respectively that constitute job J;
- **MapDurations (M^J):** the array consisting of N_m^J durations of map tasks.
- **FirstShuffleDurations (Sh_1^J):** the array representing durations of non-overlapping part of first shuffle tasks.
- **TypicalShuffleDurations (Sh_{typ}^J):** the array representing the durations of the typical shuffle tasks.
- **ReduceDurations (R^J):** the array consisting of N_R^J durations of reduce tasks.

4.2.2 Simulator Engine

Simulator Engine is the main component of SimMR which replays the given job trace. It manages all the discrete events in simulated time and performs the appropriate action on each event. It maintains data structures similar to the Hadoop job master such as a queue of submitted jobs *jobQ*. The slot allocation algorithm makes a new decision when a map or reduce task completes. Since our goal is to be fast and accurate, we simulate the jobs at the task level and do not simulate details of the TaskTrackers.

The simulator engine reads the job trace from the trace database. It communicates with the scheduler policies using a very narrow interface consisting of the following functions:

1. chooseNextMapTask(jobQ),
2. chooseNextReduceTask(jobQ)

These two functions ask the scheduling policy to return the jobId of the job whose map (or reduce) task should be executed next.

The simulator maintains a priority queue Q for *seven event types*: job arrivals and departures, map and reduce task arrivals and departures, and

an event signaling the completion of the map stage. Each event is a triplet $(eventTime, eventType, jobId)$ where $eventTime$ is the time at which the event will occur in the simulation; $eventType$ is one of the seven event types; and $jobId$ is the job index of the job with which the event is associated.

The simulator engine fires events and runs the corresponding event handlers. It tracks the number of completed map and reduce tasks and the number of free slots. It allocates the map slots to tasks as dictated by the scheduling policy. When $minMapPercentCompleted$ percentage of map tasks are completed (it is the parameter set by the user), it starts scheduling reduce tasks. We could have started the reduce tasks directly after the map stage is complete. However, the shuffle phase of the reduce task occupies a reduce slot and has to be modeled as such. Hence, we schedule a filler reduce task of infinite duration and update its duration to the first shuffle duration when all the map tasks are complete. This enables accurate modeling of the shuffle phase.

4.2.3 Scheduling policies

Different scheduling and resource allocation policies can be used with SimMR for their evaluation, e.g.:

- **FIFO:** This policy finds the earliest arriving job that needs a map (or reduce) task to be executed next.
- **MaxEDF:** Similar to FIFO, this policy finds the job with the earliest deadline which has an unscheduled map (or reduce) task.
- **MinEDF:** This policy calculates the minimum number of map and reduce slots that need to be allocated for the job to be completed within the user specified deadline, when the job arrives into the system as described later in Section 4.4. It also keeps track of the number of running and scheduled map and reduce tasks so that they are always less than the "wanted" number of slots.

4.3 SimMR Evaluation

In this section, we evaluate the accuracy and performance of SimMR, and compare it against the open source, Apache's MapReduce simulator, called Mumak [31].

4.3.1 Mumak and Rumen

Rumen [32] is a data extraction and analysis tool built for MapReduce environments. Rumen (similar to our MRProfiler) can process job history logs to generate trace files describing the task durations, the number of bytes and records read and written, etc. The trace files generated by Rumen can be replayed by the MapReduce simulator Mumak [31]. Rumen collects more than 40 properties for each map/reduce task and all the job counters. On the other hand, our MRProfiler is selective and stores only the task durations. However, MRProfiler is easily extendable if we find that additional job metrics are needed for the simulation.

An overarching design goal for Mumak is that it aims to execute the exact same MapReduce schedulers "as-is" without any changes. SimMR, on the other hand, does not have this objective and interfaces with the scheduling policy using a very narrow interface. Mumak does not simulate the running of the actual map/reduce tasks. Similar to SimMR, Mumak uses a special AllMapsFinished event generated by the SimulatedJobTracker to trigger the start of the reduce-phase. However, Mumak models the total runtime of the reduce task as the summation of the time taken for completion of all maps and the time taken for an individual task to complete the reduce phase (without the shuffle). Thus, Mumak does not model the shuffle phase accurately.

4.3.2 Experimental Testbed

We perform our experiments on 66 HP DL145 GL3 machines. Each machine has four AMD 2.39MHz cores, 8 GB RAM and two 160GB hard disks. The machines are set up in two racks and interconnected with gigabit Ethernet. We used Hadoop 0.20.2 with two machines for JobTracker and NameNode, and remaining 64 machines as worker nodes. Each slave is configured with a single map and reduce slot. The default blocksize of the file system is set to

64MB and the replication level is set to 3. We disabled speculation as it did not lead to any significant improvements.

4.3.3 Workload Trace

Our workload consists of a set of representative applications executed on three different datasets as follows:

1. **Word count:** This application computes the occurrence frequency of each word in 32GB, 40GB and 43GB Wikipedia article history dataset.

2. **Sort:** The Sort application sorts 16GB, 32GB and 64GB of random data generated using random text writer in GridMix2[1].

3. **Bayesian classification:** We use a step from the example of Bayesian classification trainer in Mahout[2]. The mapper extracts features from the input corpus and outputs the labels along with a normalized count of the labels. The reduce performs a simple addition of the counts and is also used as the combiner. The input dataset is the same Wikipedia article history dataset, except the chunks are split at page boundaries.

4. **TF-IDF:** The Term Frequency - Inverse Document Frequency application is often used in information retrieval and text mining. It is a statistical measure to evaluate how important a word is to a document. We used the TF-IDF example from Mahout and used the same Wikipedia articles history dataset.

5. **WikiTrends:** We use the data from Trending Topics (TT)[3]: Wikipedia article traffic logs that were collected (and compressed) every hour in April, May and June 2010. Our MapReduce application counts the number of times each article has been visited.

6. **Twitter:** This application uses the 12GB, 18GB and 25GB twitter dataset created by Kwak et. al. [23] containing an edgelist of twitter userids. Each edge (i, j) means that user i follows user j. The Twitter

[1] http://hadoop.apache.org/mapreduce/docs/current/gridmix.html
[2] http://http://mahout.apache.org/
[3] http://trendingtopics.org

application counts the number of asymmetric links in the dataset, that is, $(i, j) \in E$, but $(j, i) \notin E$.

4.3.4 SimMR Accuracy

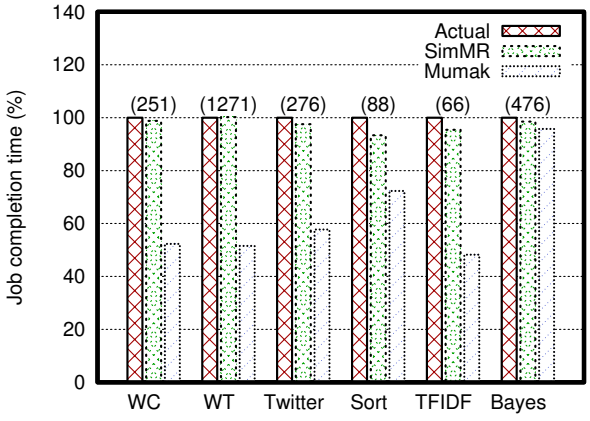
First, we evaluate the accuracy of SimMR and compare it against Mumak using the FIFO scheduler (this scheduler is available in both simulators). We collected a real workload trace consisting of three executions of the six applications. We replay this trace using SimMR and Mumak. Figure 4.2(a) shows a comparison of the duration of the simulated job with respect to the real job duration across different applications. We observe that SimMR faithfully replays the traces with less than 2.7% average (6.6% maximum) error across all the applications. On the other hand, Mumak underestimates the job completion time and has 37% average (51.7% maximum) error while replaying the same traces.

Additionally, we validate the accuracy of SimMR by simulating MinEDF and MaxEDF schedulers (discussed in more detail in the next Section 4.4) and comparing the simulation results to the testbed runs respectively. We collected a real workload trace consisting of three executions of the six applications using both the schedulers. Figures 4.2(b) and 4.2(c) show a comparison of the duration of the simulated job with respect to the real job duration across different applications. We observe that SimMR replays the traces with less than 3.7% average (8.6% maximum) error across all the applications for MaxEDF and less than 1.1% average (2.7% maximum) error for MinEDF.

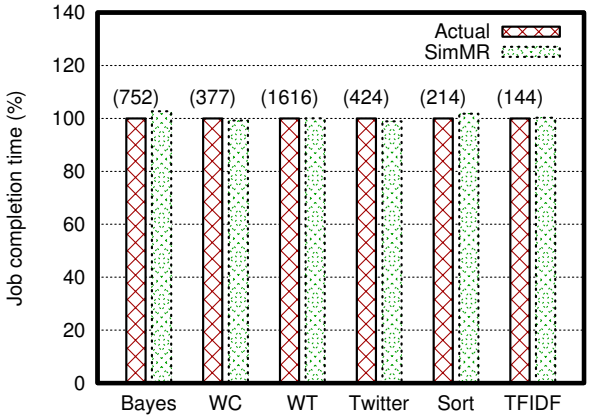
In summary, for considered diverse applications and different schedulers, SimMR replays traces with high fidelity.

4.3.5 SimMR Performance

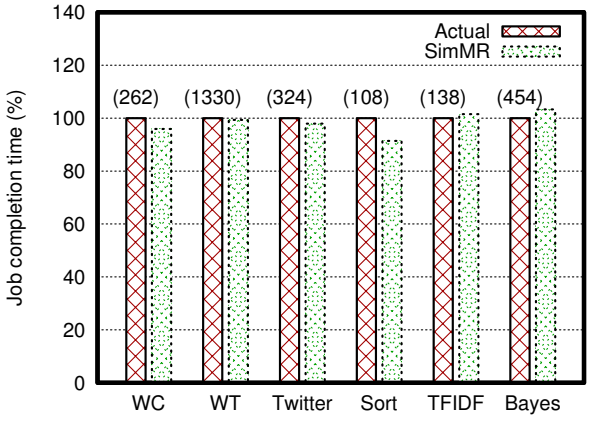
We collected traces from 1148 jobs run on our 66 node cluster during 6 months of November 2010 to April 2011. We created a single trace file (without inactivity periods) and replayed it using SimMR and Mumak. These jobs would take about a week (152 hours) if they were to be executed serially. Figure 4.3 shows the performance comparison of the two simulators. Note, that Y-axes are in log scale. SimMR replays these jobs in 1.5 seconds, while



(a) FIFO



(b) MinEDF



(c) MaxEDF

Figure 4.2: Simulator accuracy across different scheduling policies. The numbers in the parentheses above the bars represent the actual job completion time in seconds.

Mumak takes 680 seconds to replay the same set of jobs. Thus, SimMR is more than 450 times faster than Mumak in simulating these traces. On closer inspection, we observe that Mumak simulates the TaskTrackers and the heartbeats between them, which leads to greater number of simulated events and computation.

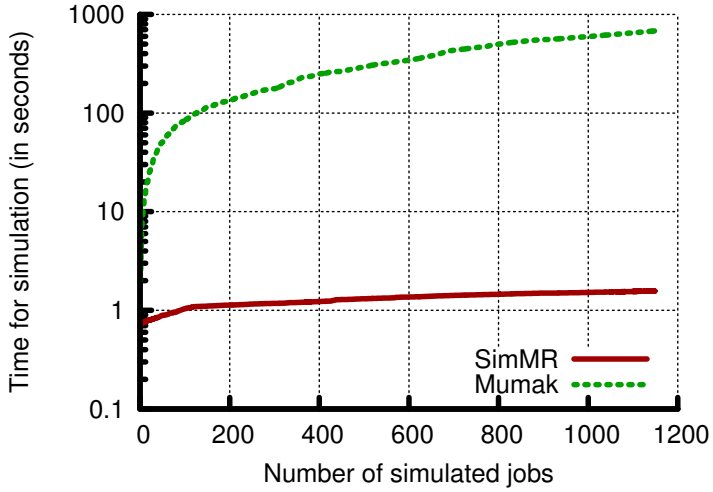


Figure 4.3: Performance comparison of simulators.

4.4 Case Study: Comparing Two Schedulers

In this section, we introduce two different deadline-based schedulers and demonstrate how to use SimMR for comparing them. Originally, Hadoop employed a simple FIFO job scheduler for efficient batch processing. Later, the Capacity scheduler [27] and Hadoop Fair Scheduler [20] were introduced for enabling multiple concurrent job executions on the same Hadoop cluster. However, these schedulers do not aim at a tailored control of allocated resources to achieve the application performance goals, e.g., the job completion time within a given (soft) deadline.

4.4.1 Deadline-based Scheduling: MaxEDF and MinEDF

The *deadline-based* scheduler should answer two inter-related questions: which job should the slots be allocated and how many slots should be allocated to

the job? In this work, we consider two policies: *MaxEDF* and *MinEDF*.

Both schedulers execute the *Earliest Deadline First* algorithm (EDF) for job ordering since this real-time scheduling is known to maximize the utility function of all the users. The difference comes in the amount of resources allocated to the job by these schedulers.

The *MaxEDF* scheduler aims to allocate the maximum available number of map (or reduce) slots for each job in the queue (i.e., apart from the EDF job ordering, the resource allocation per job is the same as under the FIFO policy). In such a way, the job might finish much earlier than the given deadline. Intuitively, such an approach optimizes pipelining of map and reduce stages of different jobs as follows. Once a map stage of a first job is completed and it proceeds to the reduce stage execution, the next job could start processing its map stage, etc. Typically, such a pipelined execution might result in the best makespan (completion time) for a given set of jobs. However, the possible drawback of the proposed schema might be that in many cases, it is impossible to preempt the already allocated resources to the earlier job (without killing its currently running tasks) to provide resources for a newly arrived job with a more "urgent" deadline.

The *MinEDF* scheduler allocates the minimal amount of map and reduce slots that would be required for meeting a given job deadline. So, this approach aims to allocate the minimum sufficient resources to the job for completing within the deadline and leaves the remaining, spare resources to the next arriving job. This minimal amount of resources is computed with a specially designed performance model introduced in [30] and briefly described below.

The proposed MapReduce performance model evaluates lower and upper bounds on the job completion time. It is based on a general model for computing performance bounds on a makespan of a given set of n tasks that are processed by k servers (e.g., n map tasks are processed by k slots in MapReduce environment). Let $T_1, T_2, \ldots, T_n$ be the duration of n tasks in a given set. Let k be the number of slots that can each execute one task at a time. The assignment of tasks to slots is done using an online, *greedy* algorithm: assign each task to the slot with the earliest finishing time. Let avg and max be the *average* and *maximum* duration of the n tasks respectively. Then the makespan of a greedy task assignment is at least $(n \cdot avg)/k$ and at most $(n-1) \cdot avg/k + max$. These lower and upper bounds

on the completion time can be easily computed if we know the average and maximum durations of the set of tasks and the number of allocated slots.

As motivated by the above model, in order to approximate the overall completion time of a MapReduce job, we need to estimate the *average* and *maximum* task durations during different execution phases of the job, i.e., map, shuffle/sort, and reduce phases. The MRProfiler (described in Section 4.2) creates the detailed job template which characterizes the task durations during all the phases of the job execution. This data is used to compute average and maximum task durations in different phases, and then to compute lower and upper bounds for each execution phase of the job. By applying the bounds model, we can express the estimates for job completion time (lower bound T_J^{low} and upper bound T_J^{up}) as a function of map/reduce tasks (N_M^J, N_R^J) and the allocated map/reduce slots (S_M^J, S_R^J) using the following equation form:

$$T_J^{low} = A_J^{low} \cdot \frac{N_M^J}{S_M^J} + B_J^{low} \cdot \frac{N_R^J}{S_R^J} + C_J^{low} \tag{4.1}$$

The equation for T_J^{up} can be written similarly (for details, see [30]). Typically, the average of lower and upper bounds is a good approximation of the job completion time.

Note, that once we have a technique for predicting the job completion time, it also can be used for solving the inverse problem: finding the appropriate number of map and reduce slots that could support a given job deadline. Equation 4.1 yields a hyperbola if S_M^J and S_R^J are the variables. All integral points on this hyperbola are possible allocations of map and reduce slots which result in meeting the same deadline. There is a point where the sum of the required map and reduce slots is minimized. We calculate this minima on the curve using Lagrange's multipliers [30], since we would like to conserve (minimize) the number of map and reduce slots required for the adequate resource allocation per job.

In such a way, the *MinEDF* scheduler allocates the minimal amount of map and reduce slots that would be needed for meeting a given job deadline.

In the simulations and the respective testbed evaluations, we will assess the quality of scheduling and resource allocation decisions by observing the following metric. Let the execution consist of a given set of n jobs $J_1, J_2, \ldots, J_n$ with corresponding deadlines $D_1, D_2, \ldots, D_n$. Let these jobs be completed at

times $T_1, T_2, \ldots, T_n$, and let Θ be the set of all jobs whose deadline has been exceeded. Then we compute the following utility function: $\sum_{J \in \Theta} \frac{T_J - D_J}{D_J}$ This function denotes the sum of *relative deadlines exceeded*. The scheduling and resource allocation policy that minimizes this value is a better candidate for a deadline-based scheduler.

To compare performance of *MaxEDF* and *MinEDF*, we analyze these policies with our simulator SimMR and the following workloads:

1. A real testbed trace comprised of multiple job runs in our 66-node cluster, and

2. A synthetic trace generated with statistical distributions that characterize the Facebook workload.

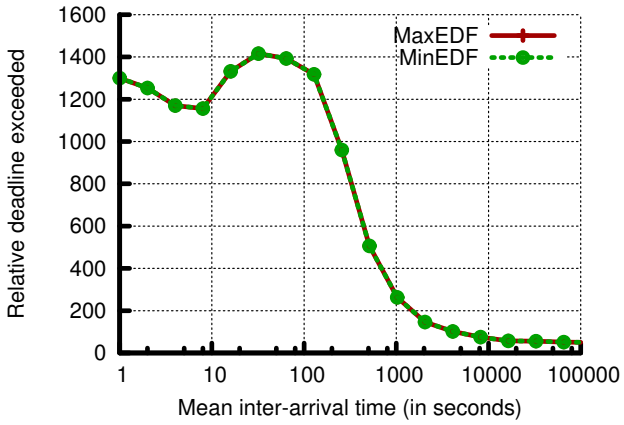
4.4.2 Replaying Real Traces with SimMR

For the real workload trace, we use a mix of the six realistic applications with different input dataset sizes as introduced and described in Section 4.3.3. We run these applications with three different datasets in our 66-nodes Hadoop testbed, and then by using MRProfiler, create the replayable job traces for SimMR. We generate an equally probable random permutation of arrival of these jobs and assume that the inter-arrival time of the jobs is exponential.

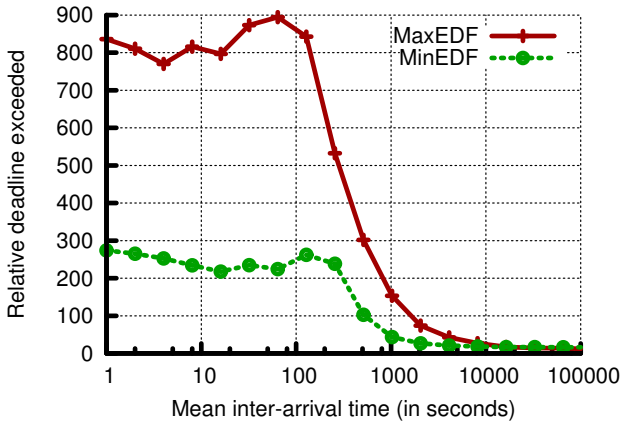
The job deadline (which is relative to the job completion time) is set to be uniformly distributed in the following interval $[T_J, df \cdot T_J]$, where T_J is the completion time of job J given all the cluster resources (i.e., maximum amount of map/reduce slots that job can utilize) and where $df \geq 1$ is a given deadline factor.

We run the simulation 400 times and report the average deadline exceeded metric while varying the mean of the exponential inter arrival times and the deadline factor as shown in Figure 4.4.

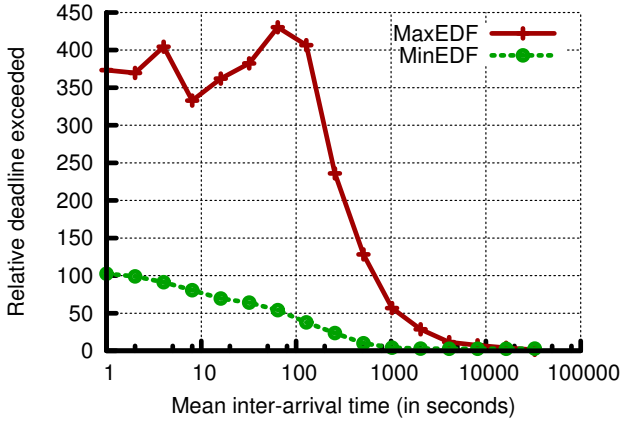
When the deadline factor is set to 1, i.e., $df = 1$, then the performance of MinEDF and MaxEDF policies coincide as shown in Figure 4.4(a), because the maximum amount of map/reduce slots that job can utilize should be allocated under both policies. The relative deadline exceeded metric decreases as the arrival rate of jobs decreases and a larger fraction of them is capable of meeting the targeted deadline. There is a slight "bump" around the mean arrival time of 100s. On closer inspection we found that this is caused



(a) Deadline factor = 1



(b) Deadline factor = 1.5



(c) Deadline factor = 3

Figure 4.4: Simulating *MaxEDF* and *MinEDF* with Real Testbed Workload.

because the scheduler does not pre-empt tasks themselves. So, if a decision to allocate resources to a task has been made the slot is not available for allocation to the earlier deadline job which just arrived.

When the deadlines are relaxed by a factor of 1.5, MinEDF allocates the minimum required slots and shares the cluster resources among multiple jobs more efficiently. This leads to a smaller value of relative deadline exceeded metric as shown in Figure 4.4(b). The performance gap between MinEDF and MaxEDF policies increases when the deadline is further relaxed by a factor of 3 as shown in Figure 4.4(c).

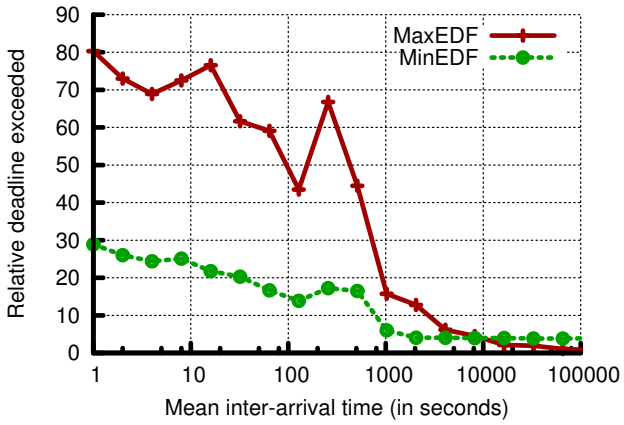
In summary, for the realistic testbed workload and a variety of studied parameters, the MinEDF scheduler shows superior results compared to the MaxEDF policy.

4.4.3 Replaying Synthetically Generated Facebook Trace

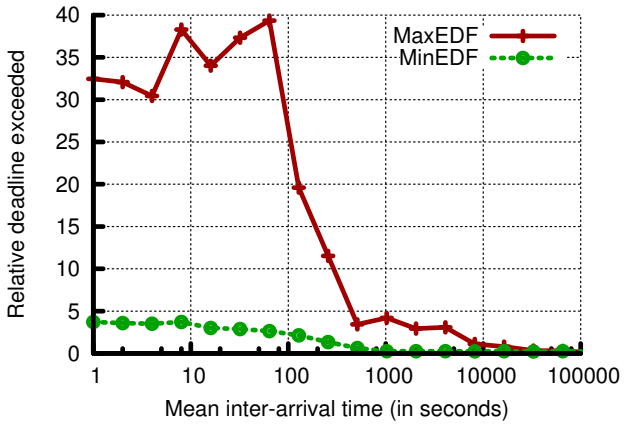
In this section, we extend the performance comparison of MinEDF and MaxEDF policies by using SimMR and a set of synthetically generated traces. Zaharia et. al. [20] provide a detailed description about MapReduce jobs in production at Facebook in October 2009 (we use Figure 1 and Table 3 from [20]). We extract the CDF from the plot of map and reduce durations in Figure 1 of [20], and then we try to identify the statistical distributions which best fits the provided plot. We fit more than 60 distributions such as Weibull, LogNormal, Pearson, Exponential, Gamma, etc. using StatAssist[4]. Our analysis shows that the LogNormal distribution fits best the provided CDF of the Facebook task duration distribution. $LN(9.9511, 1.6764)$ fits the map task CDF with a Kolmogorov-Smirnov value of 0.1056, where $LN(x; \mu, \sigma) = \frac{1}{x\sigma\sqrt{2\pi}} e^{-\frac{(\ln x - \mu)^2}{2\sigma^2}}$ is the Log-normal distribution with mean μ and variance σ. For the reduce task duration, $LN(12.375, 1.6262)$ fits with a Kolmogorov-Smirnov value of 0.0451.

In our Synthetic TraceGen module, we use these respective LogNormal distributions to generate a synthetic workload that is similar to a reported Facebook workload. Figure 4.5 shows the SimMR's outcome of replaying the generated synthetic workloads with MinEDF and MaxEDF policies. The performance results are consistent with the outcome of testbed traces' simula-

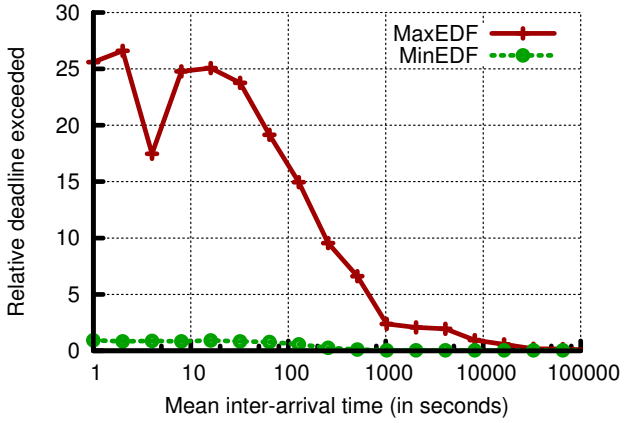
[4]http://www.mathwave.com/help/easyfit/html/tools/assist.html



(a) Deadline factor = 1.1



(b) Deadline factor = 1.5



(c) Deadline factor = 2

Figure 4.5: Simulating *MaxEDF* and *MinEDF* with Synthetic Facebook Workload.

tions: the MinEDF scheduler significantly outperforms the MaxEDF policy.

4.5 Deadline-based Workload Management

In this section, we introduce and analyze a set of complementary mechanisms that enhance workload management decisions for processing MapReduce jobs with deadlines. The three mechanisms we consider are the following:

1) *An ordering policy for the jobs in the processing queue.* For example, even if no profiling information is available about the arriving MapReduce jobs, one can utilize the job deadlines for ordering. The job ordering based on the EDF policy (*Earliest Deadline First*) was successfully used in real-time processing. The EDF job ordering might be used with a default resource allocation policy in Hadoop, where the maximum number of available map (or reduce) slots is allocated to each job at the head of the queue.

2) *A mechanism for allocating a tailored number of map and reduce slots* to each job for supporting the job completion goals. If the job profiling information is available, then our resource allocation policy can be much more precise and intelligent: for each job with a specified deadline, we can estimate and allocate the appropriate number of map and reduce slots required for completing the job within the deadline. The interesting feature of this mechanism is that as time progresses and the job deadline gets closer, the introduced mechanism can recompute (and adjust) the amount of resources needed by each job to meet its deadline.

3) *A mechanism for allocating and deallocating spare resources* in the system among active jobs. Assume that a cluster has spare resources, i.e., unallocated map and reduce slots left after each job was assigned its minimum resource quota for meeting a given deadline. It is beneficial to design a mechanism that allocates these spare resources among the running jobs to improve the Hadoop cluster utilization and its performance. The main challenge in designing such a mechanism is accurate decision making on how the slots in the cluster should be re-allocated or de-allocated to the newly-arrived job with an earlier deadline. The naïve approach could de-allocate the spare resources by canceling their running tasks. However, it may lead to undesirable churn in resource allocation and wasted, unproductive usage of cluster resources. We introduce a novel mechanism that enables a sched-

uler to accurately predict whether the cluster will have a sufficient amount of released resources over time for the new job to be completed within its deadline. The mechanism exploits the job profile information for making the prediction. It uses a novel modeling technique to avoid canceling the currently running tasks if possible. The mechanism de-allocates the spare slots only when the amount of released resources over time does not guarantee a timely completion of the newly arrived job.

We analyze the functionality and performance benefits of each mechanism via an extensive set of simulations over diverse workload sets. The solution that integrates all the three mechanisms is a clear winner in providing the most efficient support for serving MapReduce jobs with deadlines. We observe similarity in simulation results for two different workload sets, that leads us to believe in the generality of presented conclusions. The results of our simulation study are validated on a 66-node Hadoop cluster.

4.6 Three Pieces of the Workload Management Puzzle

In this section, we introduce three complementary mechanisms that enhance the scheduling and resource allocation decisions for processing MapReduce jobs with deadlines.

4.6.1 Job Ordering Policy

Job ordering in workload management emphasizes solely the ordering of jobs to achieve performance enhancements. For example, real-time operating systems employ a dynamic scheduling policy called Earliest Deadline First (EDF) which is one of traditional (textbook) scheduling policies for jobs with deadlines. The nature of MapReduce job processing differs significantly from the traditional EDF assumptions. None of the known classic results are directly applicable to job/task scheduling with deadlines in MapReduce environments Therefore, the use of EDF job ordering as a basic mechanism for deadline-based scheduling in MapReduce environments will not alone be sufficient to support the job completion time guarantees.

4.6.2 Resource Allocation Policy

Job scheduling in Hadoop is performed by a master node. Job ordering defines which job should be processed next by the master. In addition, the scheduling policy of the job master should decide how many map/reduce slots should be allocated to a current job.

The *default resource allocation policy* in Hadoop assigns the *maximum* number of map (or reduce) slots for each job in the queue. We denote Earliest Deadline First job ordering that operates with a default resource allocation as just *EDF*. This policy reflects the performance that can be achieved when there is no additional knowledge about performance characteristics of the arriving MapReduce jobs. However, the possible drawback of this default policy is that it always allocates the maximum resources to each job, and does not try to tailor the appropriate amount of resources that is necessary for completing the job within its deadline. Therefore, in many cases, it is impossible to preempt/reassign the already allocated resources (without killing the running tasks) to provide resources for a newly arrived job with an earlier deadline.

If job profiles are known, we can use this additional knowledge in performance modeling for the accurate estimates of map and reduce slots required for completing the job within the deadline. We call the mechanism that allocates the minimal resource quota required for meeting a given job deadline as *MinEDF*. The interesting and powerful feature of this mechanism is that as the time progresses and the job deadline gets closer to the current time, the introduced mechanism can recompute and adjust the amount of resources needed to each job to meet its deadline.

4.6.3 Allocating and De-allocating Spare Cluster Resources

When there is a large number of jobs competing for cluster resources the mechanism that allocates only the minimal quota of map and reduce slots for meeting job deadlines is appealing and may seem like the right approach. However, assume that a cluster has spare resources, i.e., unallocated map and reduce slots left after each job has been assigned its minimum resource quota. Then, the question is whether we could design a mechanism that allocates these spare resources among the currently active jobs to improve

Algorithm 3 MinEDF-WC Algorithm

Input: New Job $\hat{J}$ with deadline $D_{\hat{J}}$, Priority Queue of currently executing *jobs*, Number of free map slots F_M, Number of free reduce slots F_R, Number N_M^j of map tasks and N_R^j of reduce tasks in Job j

On the arrival of new job $\hat{J}$:

(MinMaps$_{\hat{J}}$, MinReduces$_{\hat{J}}$) $\leftarrow$ **ComputeMinResources**($\hat{J}$, $D_{\hat{J}}$)

// Do we have enough resources to meet this job's deadline right now?

if MinMaps$_{\hat{J}} < F_M$ **and** MinReduces$_{\hat{J}} < F_R$ **then return**

// Will we have enough resources in the future?

Sort *jobs* by increasing task durations

for each job j **in** *jobs* **do**

  if CompletedMaps$_j < N_M^j$ **and** MinMaps$_{\hat{J}} > F_M$ **then**

    // Job j is in the Map stage and $\hat{J}$ is short on map slots

    ExtraMaps$_j \leftarrow$ RunningMaps$_j -$ MinMaps$_j$

    $F_M \leftarrow F_M +$ ExtraMaps$_j$

    (MinMaps$_{\hat{J}}$, MinReduces$_{\hat{J}}$) $\leftarrow$ **ComputeMinResources**($\hat{J}$, $D_{\hat{J}}$ - M_{avg}^j)

    if MinMaps$_{\hat{J}} < F_M$ **and** MinReduces$_{\hat{J}} < F_R$ **then return**

  else if CompletedMaps$_j = N_M^j$ **and** MinReduces$_j > F_R$ **then**

    // Job j is in the Reduce stage and $\hat{J}$ is short on reduce slots

    ExtraReduces$_j \leftarrow$ RunningReduces$_j -$ MinReduces$_j$

    $F_R \leftarrow F_R +$ ExtraReduces$_j$

    (MinMaps$_{\hat{J}}$, MinReduces$_{\hat{J}}$) $\leftarrow$ **ComputeMinResources**($\hat{J}$, $D_{\hat{J}}$ - R_{avg}^j)

    if MinMaps$_{\hat{J}} < F_M$ **and** MinReduces$_{\hat{J}} < F_R$ **then return**

  end if

end for

// Not enough resources to meet deadline in future, need to kill tasks

for each job j in *jobs* **do**

  if RunningMaps$_j >$ MinMaps$_j$ **then**

    $F_M \leftarrow F_M +$ RunningMaps$_j -$ MinMaps$_j$

    KillMapTasks(j, RunningMaps$_j -$ MinMaps$_j$)

    if MinMaps$_{\hat{J}} < F_M$ **then return**

  end if

end for

On release of a map slot:

Find job j among *jobs* with earliest deadline **and** CompletedMaps$_j < N_M^j$ **and** RunningMaps$_j <$ MinMaps$_j$ **return** j

if such job j is not found, **then return** job j with the earliest deadline with CompletedMaps$_j < N_M^j$

On release of a reduce slot:

Find job j among *jobs* with earliest deadline **and** CompletedMaps$_j = N_M^j$ **and** CompletedReduces$_j < N_R^j$ **and** RunningReduces$_j <$ MinReduces$_j$ **return** j

if such job j is not found, **then return** job j with the earliest deadline with CompletedMaps$_j = N_M^j$ **and** CompletedReduces$_j < N_R^j$

the Hadoop cluster utilization and its performance, but in case of a new job arrival with an earlier deadline, these slots can be dynamically de-allocated

(if necessary) to service the newly-arrived job with an earlier deadline.

Extracted job profiles can be used for two complementary goals. First, they can be used for performance modeling of the job completion time and required resources. Second, since the job profiles provide information about map/reduce task durations, these metrics can be used to estimate when the allocated map/reduce slots are going to be released back to the job master for re-assignment. This way, we have a powerful modeling mechanism that enables a scheduler to predict whether the newly arrived job (with a deadline earlier than deadlines of some actively running jobs) can be completed in time by simply waiting while some of the allocated slots finish processing their current tasks before being re-allocated to the newly arrived job. If the prediction returns a negative answer, i.e., the amount of released resources over time does not guarantee a completion of the newly arrived job with a given deadline, then the scheduler makes a decision of how many of the slots (as well as which ones) should cancel processing their tasks, and be re-allocated to the new job immediately. For this cancellation procedure, the scheduler only considers spare slots, i.e., the slots that do not belong to the minimum quota of slots allocated to currently running jobs for meeting their deadlines.

This new mechanism further enhances the *MinEDF* functionality to efficiently utilize spare cluster resources. It resembles *work-conserving scheduling*, so we call it as *MinEDF-WC*. We have implemented *MinEDF-WC* as a novel deadline-based Hadoop scheduler that integrates all three mechanisms to provide an efficient support for MapReduce jobs with deadlines. A pseudo-code shown in Algorithm 3 presents job scheduling and slot allocation scheme.

4.7 Summary

To become enterprise-ready, the Hadoop open-source stack needs to be enhanced with new tools required in enterprise environments to support robust performance management. Due to lack of performance guarantees for job completion times when executed in shared environments (while many enterprise applications require such time guarantees), there is a need for new workload management strategies and supporting tools. In this work, we intro-

duce a simulation environment SimMR that can assist system administrators in performance analysis and evaluation of new resource allocation and job scheduling algorithms in large-scale distributed systems such as Hadoop and other performance related tasks in MapReduce cluster management. The proposed SimMR simulator is accurate and fast: it can simulate a complex multi-hour workload in less than a second. It is aimed at helping Hadoop administrators in their daily tasks: SimMR can quickly replay production cluster workloads with different scenarios of interest, assess various *what-if* questions, and help avoiding error-prone decisions.

Chapter 5

Minimizing Makespan of Set of MapReduce Jobs

5.1 Motivation

Job execution efficiency is important for processing production workloads when a given set of MapReduce jobs and workflows needs to be executed periodically on new data. Typically, the default FIFO scheduler is used for processing production jobs since the primary performance objective is to *minimize the overall execution time (makespan)* of a given set. Such production workloads are analyzed off-line for optimizing their execution. To ease the task of writing complex analytics programs, high-level SQL-like abstractions such as Pig and Hive have been proposed. There is a slew of optimization methods introduced for improving data read/write efficiency in a set of production jobs. For different MapReduce jobs operating over the same dataset, a more efficient job scheduling [33] proposes merge their executions so that the input data is only scanned once.

In this chapter, we consider a subset of a production workload formed by the jobs with no dependencies. Such independent jobs arise, for example, while processing different datasets, or optimized Pig/Hive queries resulting in a single MapReduce job[1]. We discuss a different cause for a job execution inefficiency inherent to the MapReduce computation that processes map and reduce tasks separated by a synchronization barrier. The order in which jobs are executed can have a significant impact on the overall processing time, and therefore, on the achieved cluster utilization. For data-dependent jobs, the successive job can only start after the current one is entirely finished. However, for data-independent jobs, once the previous job completes its map stage and begins the reduce stage, the next job can start executing its map

[1] 11 out of 17 PigMix queries (http://wiki.apache.org/pig/PigMix) translate to a single independent MapReduce job.

stage with the released map resources in a pipelined fashion. Thus, there is an overlap in job executions when different jobs use complementary cluster resources: map and reduce slots. Note that a larger overlap in job executions leads to better job pipelining, increased cluster utilization, and an improved execution time, while using the same number of machines (and thus for free).

We introduce a simple abstraction of a MapReduce job as a pair of its map and reduce stage durations. This representation enables us to apply the classic Johnson algorithm [34] that was designed for building an optimal two-stage job schedule. Since the set of production jobs is executed periodically, it permits their automated profiling from *past executions*. When jobs in a batch need to process new datasets, we use the knowledge of extracted job profiles to pre-compute new estimates of jobs' map and reduce stage durations, and then construct an optimized schedule for *future executions*. We evaluate performance benefits of the constructed schedule through extensive simulations over a variety of realistic workloads. The performance results are workload and cluster-size dependent, but we typically achieve up to 10%-25% makespan improvements.

However, the proposed abstraction obscures the amount of resources each job may be able to utilize, and in some cases, Johnson's schedule may lead to a suboptimal makespan. We design *BalancedPools*, a *novel heuristic* that efficiently utilizes characteristics and properties of MapReduce jobs in a given workload for constructing the optimized job schedule. We evaluate the performance benefits of the constructed schedule through simulations over a variety of realistic workloads. The detailed evaluation of the proposed heuristic demonstrates makespan improvements of up to 15%-38% for situations where Johnson's schedule is suboptimal. The results of our simulation study are validated through experiments on a 66-node Hadoop cluster.

5.2 Optimized Batch Scheduling

In this section, we discuss the problem of minimizing the overall completion time for a given set of MapReduce jobs. We present a simple but effective *abstraction* of the MapReduce job execution that enables us to apply the classic Johnson algorithm for building an optimized job schedule. Then we discuss possible inefficiencies of this abstraction and a novel heuristic as an

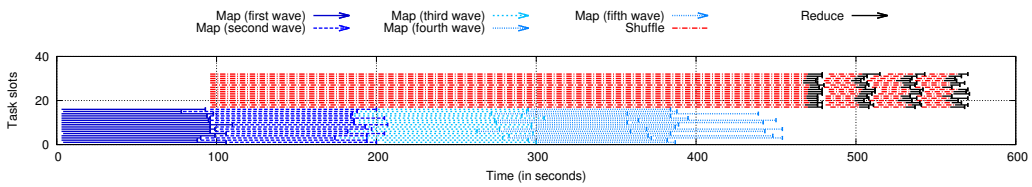


Figure 5.1: WikiTrends application executed in a Hadoop cluster with 16 map and 16 reduce slots.

alternative solution for an optimized schedule of a given set of MapReduce jobs.

5.2.1 Problem Definition

Each MapReduce job consists of a specified number of map and reduce tasks. The job execution time and specifics of the execution depend on the amount of resources (map and reduce slots) allocated to the job. Section 6.4 presents an example of Wikitrends application processing. Figure 5.1 shows a detailed visualization of how 71 map and 64 reduce tasks of this application are processed in the Hadoop cluster with 16 map and 16 reduce slots. Instead of the detailed job execution at the task level, we introduce a simple **abstraction**, where each MapReduce job J_i is defined by durations of its map and reduce stages m_i and r_i, i.e., $J_i = (m_i, r_i)$. Section 6.4 presents our profiling approach and performance model for computing the estimates of average map and reduce stage durations when the job is executed on a new dataset. This model is applied to derive the proposed new abstraction $J_i = (m_i, r_i)$.

Let us consider the execution of two (independent) MapReduce jobs J_1 and J_2 in a Hadoop cluster with a FIFO scheduler. There are no data dependencies between these jobs. Therefore, once the first job completes its map stage and begins reduce stage processing, the next job can start its map stage execution with the released map resources in a pipelined fashion (see Figure 5.2). There is an "overlap" in executions of map stage of the next job and the reduce stage of the previous one.

We note an interesting observation about the execution of such jobs. Some of the execution orders may lead to a significantly less efficient resource usage and an increased processing time. As a motivating example, let us consider two independent MapReduce jobs that utilize all the given Hadoop cluster's

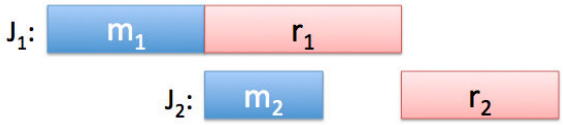
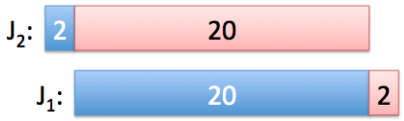


Figure 5.2: Pipelined execution of two MapReduce jobs J_1 and J_2.

resources and that result in the following map and reduce stage durations: $J_1 = (20s, 2s)$ and $J_2 = (2s, 20s)$. In the Hadoop cluster with the FIFO scheduler, they can be processed in two possible ways:

(a) J_1 is followed by J_2.



(b) J_2 is followed by J_1.

Figure 5.3: Impact of different job schedules on overall completion time.

- J_1 is followed by J_2 (as shown in Figure 5.3 (a)). The reduce stage of J_1 overlaps with the map stage of J_2 leading to overlap of only $2s$. Thus, the total completion time of processing two jobs is $20s+2s+20s = 42s$.
- J_2 is followed by J_1 (as shown in Figure 5.3 (b)). The reduce stage of J_2 overlaps with the map stage of J_1 leading to a much better pipelined execution and a larger overlap of 20s. Thus, the total makespan is $2s + 20s + 2s = 24s$.

Thus, there can be a significant difference in the overall job completion time (75% in the example above) depending on the execution order of the jobs.

We consider the *following problem*. Let $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ be a set of n MapReduce jobs with no data dependencies between them. We aim to determine an order (a schedule) of execution of jobs $J_i \in \mathcal{J}$ such that the makespan of the entire set is minimized.

5.2.2 Johnson's Algorithm

In 1953, Johnson [34] proposed an optimal algorithm for two stage production schedule. In the original problem formulation, a set of production items and two machines (S_1 and S_2) are given. Each item must pass through stage one that is served by machine S_1, and then stage two that is served by machine S_2. Each machine can handle only one item at a time. The production item i in the set is represented by two positive[2] numbers (s_i^1, s_i^2) that define service times for the item to pass through stages one and two respectively.

There is a striking similarity between the problem formulation described above and the problem that we would like to solve: building a schedule that minimizes the makespan of a given set of MapReduce jobs. We can represent each MapReduce job J_i in our batch set $\mathcal{J}$ by a pair of computed durations (m_i, r_i) of its map and reduce stages, and these stage durations fairly define the "busy" processing times by the map and reduce slots respectively. This abstraction enable us to apply Johnson's algorithm (offered for building the optimal two-stage jobs' schedule) to our scheduling problem for a set of MapReduce jobs. Now, we explain the essence of Johnson's algorithm in terms of MapReduce jobs.

Let us consider a collection $\mathcal{J}$ of n jobs, where each job J_i is represented by the pair (m_i, r_i) of map and reduce stage durations respectively. Let us augment each job $J_i = (m_i, r_i)$ with an attribute D_i that is defined as follows:

$$D_i = \begin{cases} (m_i, \texttt{m}) & \text{if } \min(m_i, r_i) = m_i, \\ (r_i, \texttt{r}) & \text{otherwise.} \end{cases}$$

The first argument in D_i is called the *stage duration* and denoted as D_i^1. The second argument is called the *stage type* (map or reduce) and denoted as D_i^2.

Algorithm 1 shows how an optimal schedule can be constructed using Johnson's algorithm. First, we sort all the n jobs from the original set $\mathcal{J}$ in the ordered list L in such a way that job J_i precedes job J_{i+1} if and only if $min(m_i, r_i) \leq min(m_{i+1}, r_{i+1})$. In other words, we sort the jobs using the stage duration attribute D_i^1 in D_i (it represents the smallest duration of the two stages). Then the algorithm works by taking jobs from list L and placing them into the schedule σ from the both ends (head and tail) and proceeding towards the middle. If the stage type in D_i is $\texttt{m}$, i.e., represents the map

[2] In fact, Johnson's schedule is also optimal for the case when $s_i^2 = 0$.

stage, then the job J_i is placed from the head of the schedule, otherwise from the tail. The complexity of Johnson's Algorithm is dominated by the sorting operation and thus is $\mathcal{O}(n \log n)$.

Algorithm 4 Johnson's Algorithm

Input: A set $\mathcal{J}$ of n MapReduce jobs. D_i is the attribute of job J_i as defined above.
Output: Schedule σ (order of jobs execution.)

1: Sort the original set $\mathcal{J}$ of jobs into the ordered list L using their stage duration attribute D_i^1
2: $head \leftarrow 1, tail \leftarrow n$
3: **for each** job J_i in L **do**
4: **if** $D_i^2 = \texttt{m}$ **then**
5: *// Put job J_i from the front*
6: $\sigma_{head} \leftarrow J_i$, head $\leftarrow$ head + 1
7: **else**
8: *// Put job J_i from the end*
9: $\sigma_{tail} \leftarrow J_i$, tail $\leftarrow$ tail - 1
10: **end if**
11: **end for**

Let us illustrate the job schedule construction with Johnson's algorithm for a simple example with five MapReduce jobs shown in Figure 5.4. These jobs are augmented with additional computed attribute D_i shown in the last column.

J_i	m_i	r_i	D_i
J_1	4	5	(4, m)
J_2	1	4	(1, m)
J_3	30	4	(4, r)
J_4	6	30	(6, m)
J_5	2	3	(2, m)

Figure 5.4: Example of five MapReduce jobs.

J_i	m_i	r_i	D_i
J_2	1	4	(1, m)
J_5	2	3	(2, m)
J_1	4	5	(4, m)
J_3	30	4	(4, r)
J_4	6	30	(6, m)

Figure 5.5: The ordered list L of five MapReduce jobs.

At first, this collection of jobs is sorted into a list L according to the attribute D_i^1 (i.e., first argument of D_i). The sorted list of jobs is shown in Figure 5.5. Then we follow Johnson's algorithm and start placing the jobs in the schedule σ from both ends toward the middle, and construct the following schedule:

- J_2 is represented by D_2=(1, m). Since D_2^2 = m then J_2 goes to the head of σ, and $\sigma = (J_2, ...)$.
- J_5 is represented by D_5=(1, m). Again, J_5 goes to the head of σ, and $\sigma = (J_2, J_5, ...)$.
- J_1 is represented by D_1=(4, m), and it goes to the head of σ, and $\sigma = (J_2, J_5, J_1, ...)$.
- J_3 is represented by D_3=(4, r). Since D_3^2 = r then J_3 goes to the tail of σ, and $\sigma = (J_2, J_5, J_1, ..., J_3)$.
- J_4 is represented by D_4=(1, m) and it goes to the head of σ, and $\sigma = (J_2, J_5, J_1, J_4, J_3)$.

Job ordering $\sigma = (J_2, J_5, J_1, J_4, J_3)$ defines Johnson's schedule for the job execution with the minimum overall makespan. For our example, the makespan of the optimal schedule is 47. The worst schedule is defined by the reverse order of the optimal one, i.e., $(J_3, J_4, J_1, J_5, J_2)$. The worst job schedule has a makespan of 78 (this is 66% increase in the makespan compared to the optimal time). Indeed, the optimal schedule may provide significant savings.

5.2.3 BalancedPools Heuristic Algorithm

While the simple abstraction for MapReduce jobs proposed in Section 5.2.2 enables us to apply the elegant Johnson algorithm for constructing the optimized job schedule, it raises the following questions about its abstraction:

- How well does this abstraction correspond to the reality of complex execution of MapReduce jobs?
- How accurate is the *computed* makespan of Johnson's schedule for estimating the *measured* makespan of a given set of MapReduce jobs?
- What are the situations where the generated Johnson schedule might lead to suboptimal results?

When a MapReduce job is represented as a pair of map and reduce stage durations, it obscures the number of tasks that comprise the job's map and reduce stages and the number of slots that process these tasks. For example,

Figure 5.1 shows how 71 map and 64 reduce tasks of Wikitrends application are processed in the Hadoop cluster with 16 map and 16 reduce slots. Note, that the last, fifth wave of the map stage has for processing only 7 tasks (71-16x4). Thus, out of 16 available map slots only 7 slots are used by the current application and the remaining 9 map slots can be immediately used for processing of the next job. Therefore, processing of the next job's map stage may start before the previous job completes its map stage. As a result, the makespan computed by the Johnson algorithm might be pessimistic compared to the real execution of the job schedule on the Hadoop cluster. To provide better estimates for the makespan of a given set of MapReduce jobs under different job schedules, we use the MapReduce simulator SimMR [35] that can faithfully replay MapReduce job traces at the tasks/slots level: the completion times of simulated jobs are within 5% of the original ones.

Let us revisit MapReduce job processing and discuss situations where Johnson's schedule might provide a suboptimal solution. Consider the set of five jobs shown in Figure 5.4 (see Section 5.2.2). Below we describe *two different scenarios* that, in spite of their differences, lead to the same job profiles and stage durations as shown in Figure 5.4. Therefore, if we apply Johnson's algorithm, it will produce the same schedule $\sigma = (J_2, J_5, J_1, J_4, J_3)$ for minimizing the makespan of this set. In both scenarios, we consider a Hadoop cluster with 30 worker nodes, each configured with a single map and single reduce slot, i.e., with 30 map and 30 reduce slots overall.

Scenario1: Let each job in the set be comprised of 30 map and 30 reduce tasks. Thus, each job utilizes either all map or all reduce slots during its processing. In this scenario, there is a perfect match between the assumptions of the classic Johnson algorithm for two-stage production system and MapReduce job processing.

Scenario2: Let jobs J_1, J_2, and J_5 be comprised of 30 map and 30 reduce tasks, and jobs J_3 and J_4 consist of 20 map and 20 reduce tasks. Figure 5.6(a) visualizes the execution of these five MapReduce jobs according to the generated Johnson schedule $\sigma = (J_2, J_5, J_1, J_4, J_3)$.

We use a different color scheme for map (blue/dark) and reduce (red/light) stages, the height of the stages reflects the amount of resources used by the jobs, the width represents the stage duration, the jobs appear at the time line as they are processed by the schedule.

While the first three jobs J_2, J_5, and J_1 utilize all map and all reduce slots during their processing, the last two jobs J_4 and J_3 only use 20 map and 20 reduce slots, and hence map stage processing of J_3 starts earlier than the map stage of J_4 is completed because there are 10 map slots available in the system. The first 10 tasks of J_3 are processed concurrently with 20 map tasks of J_4. When J_4 completes its map stage and releases 20 map slots, then the next 10 map tasks of J_3 get processed. However, this slightly modified execution leads to the same makespan of 47 time units as under *Scenario1* because processing of J_3's reduce stage can only start when the entire map stage of J_3 is finished.

We claim that Johnson's schedule for *Scenario2* described above is suboptimal, by outlining a better solution. Let us partition these five jobs into two pools with a tailored amount of cluster resources allocated to each pool:

1. *Pool1* with J_1, J_2, and J_5 (10x10 map/reduce slots);

2. *Pool2* with J_3 and J_4 (20x20 map/reduce slots).

First of all, a different amount of resources allocated to jobs in *Pool1* changes these jobs' map and reduce stage durations. Each of these jobs has 30 map and 30 reduce tasks for processing. When these 30 tasks are processed with 10 slots, the execution takes three times longer: both map and reduce stages are processed in three waves, compared with a single wave for the stage execution with 30 slots. For jobs in each pool, we apply Johnson's algorithm to generate the optimized schedules:

1. *Pool1* is processed according to $\sigma_1 = (J_2, J_5, J_1)$. This schedule results in the makespan of 39 time units;

2. *Pool2* is executed according to $\sigma_2 = (J_4, J_3)$. This schedule results in the makespan of 40 time units.

Figure 5.6(b) visualizes the job execution of these two pools. Jobs in *Pool1* and *Pool2* are processed concurrently (each set follows its own schedule). The cluster resources are partitioned between the two pools in a tailored manner. Using this approach, the overall makespan for processing these five jobs is 40 time units, that is almost 20% improvements compared to 47 time units using Johnson's schedule. This example exploits additional properties specific to MapReduce environments and the execution of MapReduce jobs.

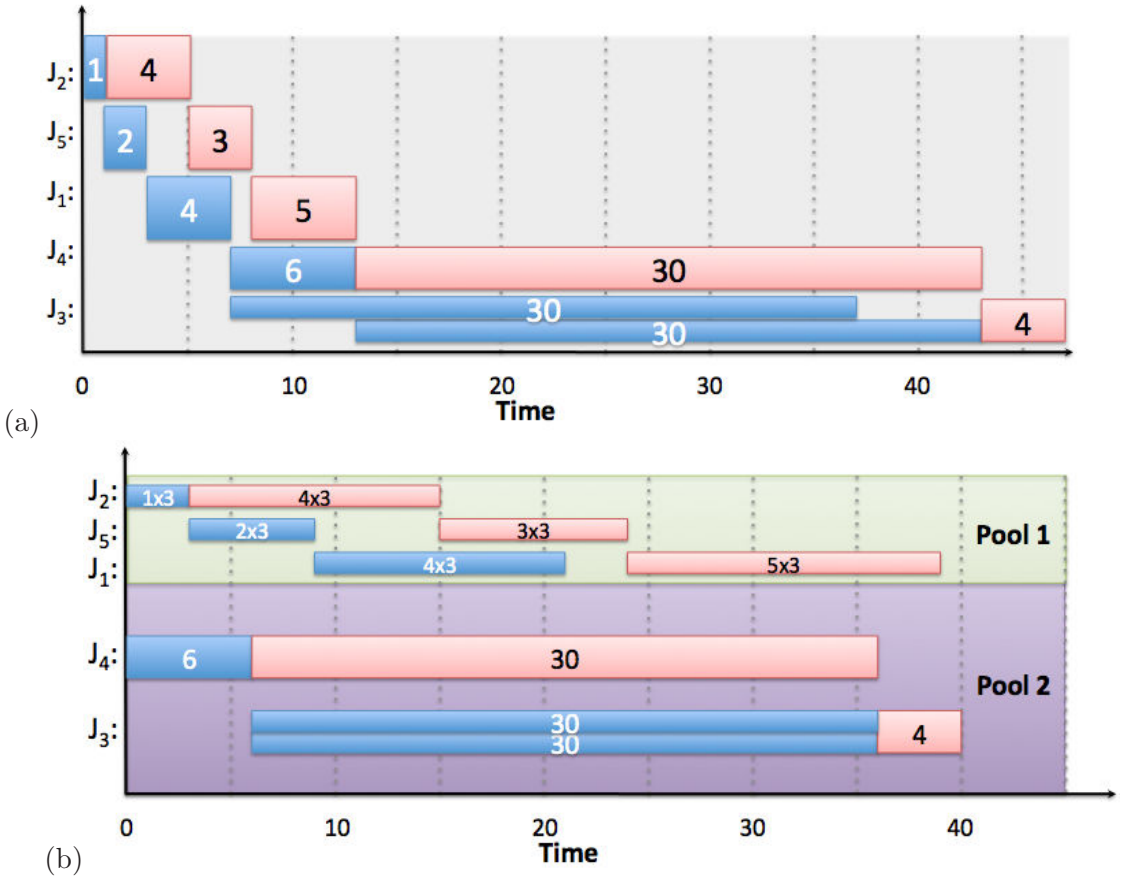


Figure 5.6: Example with five MapReduce jobs: (a) job processing with Johnson's schedule; (b) an alternative solution with BalancedPools.

In particular, the job stage durations closely depend on the amount of allocated resources (map and reduce slots). In this way, we can change the jobs' *appearance*. The main objective function of such an algorithm is to partition the jobs into two pools with specially tailored resource allocations such that the makespan of jobs in these pools are balanced, and the overall completion time of jobs in both pools is minimized. In general, the problem of balancing the map and reduce tasks in slots to achieve the minimum makespan for a set of MapReduce jobs is NP-hard. This can be easily proved by a simple polynomial reduction from the 3PARTITION problem [36]. We design a heuristic called the *BalancedPools* algorithm. As shown in Algorithm 5, we iteratively partition the jobs into two pools and then try to identify the adequate resource allocations for each pool such that the makespans of these pools are balanced. Within each pool we apply Johnson's algorithm for job

Algorithm 5 BalancedPools Algorithm

Input: 1) List J of n MapReduce jobs.
2) M: Number of machines in the cluster.
Output: Optimized Makespan

```
1: Sort J based on increasing number of map tasks
2: BestMakespan ←SIMULATE(J, JOHNSONORDER(J), M)
3: for split ← 1 to n − 1 do
4:     // Partition J into list of small Jobs_α and big Jobs_β
5:     Jobs_α ← (J_1, ··· , J_split)
6:     Jobs_β ← (J_split+1, ··· , J_n)
7:     SizeBegin ← 1, SizeEnd ← M
8:     // Binary search for the pool size that balances completion times of
       both pools
9:     repeat
10:        SizeMid ← (SizeBegin + SizeEnd)/2
11:        Makespan_α ← SIMULATE(Jobs_α,
           JOHNSONORDER(Jobs_α), SizeMid)
12:        Makespan_β ← SIMULATE(Jobs_β,
           JOHNSONORDER(Jobs_β), M - SizeMid)
13:        if Makespan_α < Makespan_β then
14:            SizeEnd ← SizeMid
15:        else
16:            SizeBegin ← SizeMid
17:        end if
18:    until SizeBegin ≠ SizeEnd
19:    Makespan ← MAX(Makespan_α, Makespan_β)
20:    if Makespan < BestMakespan then
21:        BestMakespan ← Makespan
22:    end if
23: end for
```

scheduling, where map and reduce stage durations are computed with the performance model described in Section 6.4. The pool makespan is estimated (accurately within 5%) with MapReduce simulator SimMR [35] as a part of the algorithm. The use of the simulator in the solution is absolutely necessary and justified. As we demonstrated, the makespan computation that follows Johnson's schedule and its simple abstraction may result in a significant inaccuracy, and more accurate estimates might be obtained only via MapReduce simulations at the task/slot level. The complexity of the algorithm is $\mathcal{O}(n^2 \log n \log M)$. However, SimMR can simulate a 1000 job workload on a 100 node Hadoop cluster in less than 2 seconds. The designed

algorithm can be extended to a larger number of pools at the price of a significantly higher complexity.

The job execution with two pools is implemented using Capacity scheduler [27] that allows resource partitioning into different pools with a separate job queue for each pool.

5.3 Evaluation

This section evaluates the benefits of Johnson's schedule and the novel *Balanced Pools* algorithm for minimizing the makespan of a set of MapReduce jobs using a variety of synthetic and realistic workloads derived from the Yahoo! M45 cluster. First, we evaluate the benefits of different schedules with simulation environment SimMR. Then, we validate the simulation results by performing similar experiments in a 66-node Hadoop cluster.

5.3.1 Workloads

We use the following workloads in our experiments:

1) Yahoo! M45: This workload represents a mix of 100 MapReduce jobs [3] that is based on the analysis performed on the Yahoo! M45 cluster [25], and is generated as follows:

- Each job consists of the number of map and reduce tasks drawn from the distribution $\mathcal{N}(154, 558)$ and $\mathcal{N}(19, 145)$ respectively, where $\mathcal{N}(\mu, \sigma)$ is the normal distribution with mean μ and standard deviation σ.
- Map and reduce task durations are defined by $\mathcal{N}(50, 200)$ and $\mathcal{N}(100, 300)$ respectively[4].
- To avoid that map and reduce stage durations of the jobs look similar to each other (since they are drawn from the same distribution), an additional *scale factor* is applied to map and reduce task durations of each job.

[3] We also run a mix with 10-20 jobs and obtained similar performance results.

[4] The study [25] did not report statistics of individual task durations. We use a greater range for reduce tasks since they combine shuffle, sort, reduce phase processing, and time for writing three data copies back to HDFS.

To perform a sensitivity analysis, we have created two job sets (100 jobs each) based on Yahoo! M45 workload:

1. *Unimodal* set that uses a single scale factor for the overall workload, i.e., the scale factor for each job is drawn uniformly from $[1, 10]$.

2. *Bimodal* set where a subset of jobs (80%) are scaled using a factor uniformly distributed between $[1, 2]$ and the remaining jobs (20%) are scaled using $[8, 10]$. This mimics workloads that have a large fraction of *short* jobs and a smaller subset of *long* jobs.

2) Synthetic: Additionally, we create a synthetic workload with 100 jobs having a number of map and reduce tasks drawn uniformly from $[1, 100]$ and $[1, 50]$ respectively. The map and reduce task durations are normally distributed using $\mathcal{N}(100, 1000)$ and $\mathcal{N}(200, 2000)$ respectively. We create two versions of synthetic workload: 1) *Unimodal*, where each job is scaled using a factor uniformly distributed between $[1, 10]$ and 2) *Bimodal*, where 80% of the jobs are scaled using a factor uniformly distributed between $[1, 2]$ and the remaining 20% of jobs are scaled using $[8, 10]$.

5.3.2 Simulation Results

First, we analyze the proposed job schedule algorithms and their performance using the simulation environment SimMR [35] that was designed for evaluation and analysis of different workload management strategies in MapReduce environments. SimMR can replay execution traces of real workloads collected in Hadoop clusters as well as generate and execute synthetic traces based on statistical properties of workloads. Simulating synthetic workloads is especially attractive since it enables a sensitivity analysis of scheduling policies for a variety of different MapReduce workloads.

Figure 5.7 shows the results for the synthetic workloads with *Unimodal* and *Bimodal* distributions. These graphs reflect five lines: *Min* and *Max* show theoretical makespans under Johnson's (optimal) schedule and reverse Johnson's (worst) schedule respectively. That is, if MapReduce jobs would precisely satisfy the two-stage system assumptions then the overall makespan can be easily computed from the abstraction $J_i{=}(m_i, r_i)$. A difference between *Min* and *Max* reflects achievable performance benefits under the optimal schedule for this abstraction. *MinSim* and *MaxSim* show simulated

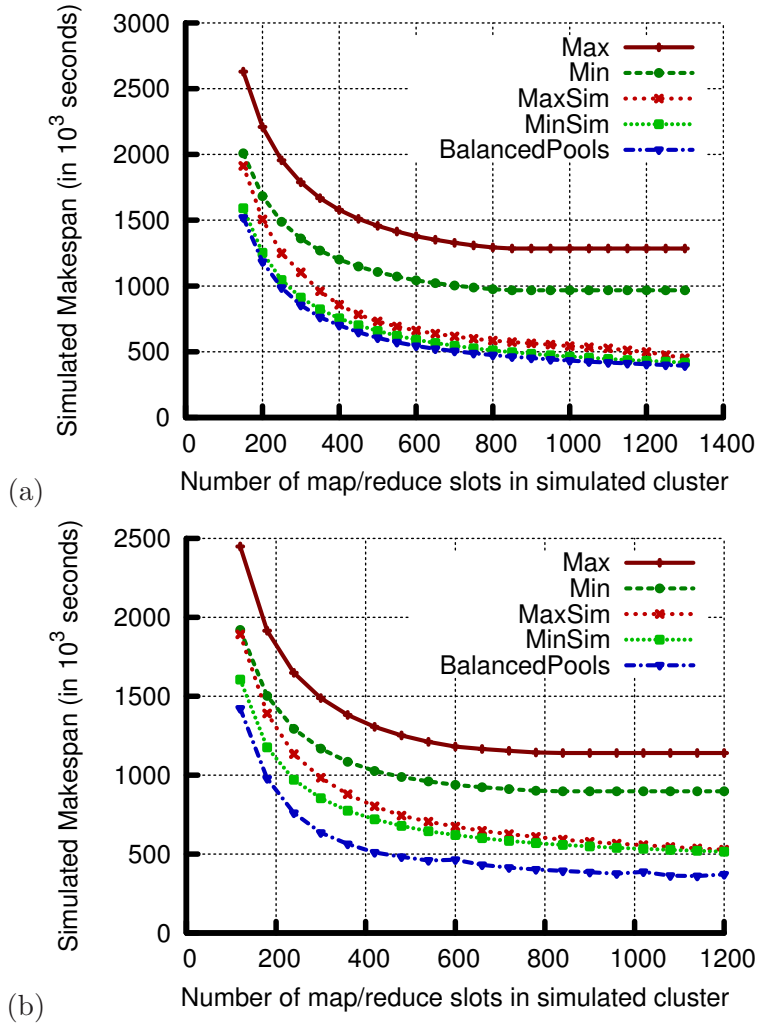


Figure 5.7: Simulating synthetic workload: (a) *Unimodal* and (b) *Bimodal*.

makespans with SimMR for a set of given MapReduce jobs under Johnson's schedule and reverse Johnson's schedule respectively. We should stress that once we consider MapReduce jobs at the tasks/slots level Johnson's schedule and reverse Johnson's schedule do not guarantee the optimal and worst makespan for this set of jobs. The difference between *MinSim* and *MaxSim* reflects a lower bound of potential benefits (since the "worst" makespan might be much worse than under *MaxSim*). Finally, *BalancedPools* reflects the simulated (with SimMR) makespan of the job schedule constructed with the new *BalancedPools* heuristic.

The X axis reflects the Hadoop cluster size (without loss of generality, we assume 1 map and 1 reduce slot per node). The algorithm performance is a

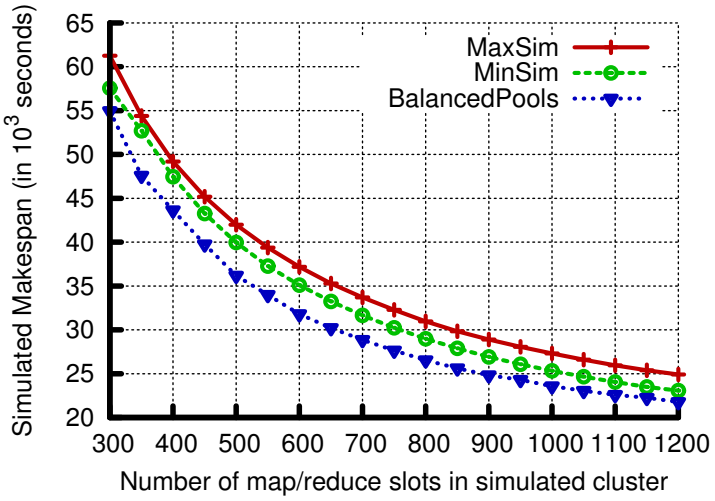
function of the cluster size: with its increase (i.e., when available resources in the cluster are plentiful), the performance benefits are diminishing as expected. However, for different workloads the points of diminishing return are different. This simulation exercise is useful for evaluating the required cluster size to support the specific (targeted) makespan for a set of given jobs.

Figure 5.7 shows that the simplified abstraction $J_i=(m_i, r_i)$ and makespan computations that use it (i.e., *Min* and *Max*) are inaccurate for estimating the real makespan of MapReduce jobs (due to lack of tasks/slots information), and in the rest of the graphs we omit these lines. This comparison strongly justifies the introduction of the simulator SimMR in the new heuristic for accurate makespan estimates.

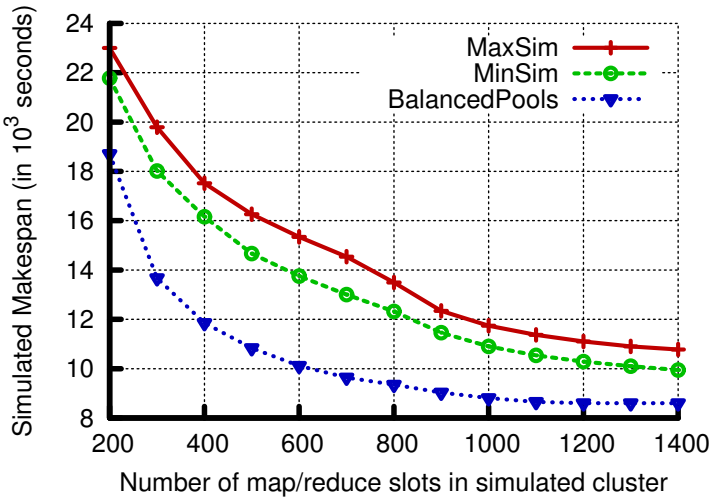
Figure 5.7(a) shows up to 25% of makespan decrease with Jonhson's schedule (*MinSim*) compared to *MaxSim* for *Unimodal* case. The benefits diminishing for larger cluster sizes. The *BalancedPools* schedule behaves similar to Johnson's in this case. However, results are very different for the *Bimodal* workload shown in Figure 5.7(b). The *BalancedPools* heuristic provides up to 38% of makespan improvements, which are much better compared to Johnson's schedule (it is suboptimal for this workload). *BalancedPools* achieves significant additional makespan improvements compared to Johnson's algorithm for a variety of different cluster sizes.

Figure 5.8 shows results of simulating the Yahoo! M45 workload (*Unimodal* and *Bimodal* types). Interestingly, Johnson's schedule provides diminished returns in both cases for Yahoo!'s workload. We can see only up to 12% of makespan improvements for most experiments. The *BalancedPools* heuristic significantly outperforms Johnson's algorithm: by 10%-30% in most cases. It shows overall makespan improvements up to 38% for the *Bimodal* Yahoo! M45 workload as shown in Figure 5.8(b).

Performance benefits under Johnson's algorithm and the *BalancedPools* heuristic are clearly workload and cluster size dependent. The proposed framework automatically constructs the optimized job schedule and provides the estimates of its makespan as a function of allocated resources. We validate the simulation results through experiments on a 66-node Hadoop cluster. These results closely follow the simulation results.



(a)



(b)

Figure 5.8: Simulating Yahoo!'s workload: (a) *Unimodal* and (b) *Bimodal*.

5.4 Summary

In this work, we considered the problem of finding a schedule that minimizes the overall completion time of a given set of independent MapReduce jobs. We designed a novel framework and a new *heuristic*, called *BalancedPools*, that efficiently utilize characteristics and properties of MapReduce jobs in a given workload for constructing the optimized job schedule.

Chapter 6

Comparing Hardware Alternatives

6.1 Motivation

Diversity of MapReduce applications creates competing requirements for performance and size of the underlying compute cluster, job scheduling, and workload management policies. While today's servers are more efficient and longer lasting, a typical data center hardware lifecycle is still around 3-5 years, and eventually, there is a need to upgrade the data center old hardware. For system administrators of MapReduce clusters, this decision point represents challenging times and inevitable efforts to analyze and compare different hardware choices as candidates for the upgrade. Similar efforts are required for assessing performance (and cost) offered by public cloud options if they are considered to complement (or replace) the capacities of internal MapReduce clusters.

Currently, most of the system administrators' efforts for performance evaluation of future hardware and its sizing are manual and guess-based. Very often, some test applications, such as *sort*, are used to compare the performance of different hardware or cluster configurations. Usually, this simple benchmarking helps in profiling the execution phases of MapReduce processing pipeline on these hardware types. However, the outcome of such a comparison is difficult to apply to other workloads of interest and use for predicting their performance on the given platforms.

There are efforts to design MapReduce benchmarks that represent a mix of synthetic jobs which mimic the job profiles mined from production loads, e.g., a family of GridMix benchmarks [37]. These benchmarks offer saturation tools for stressing the customer Hadoop clusters at scale. This approach pursues a different goal and does not have a "scaled down" version for assessing and comparing performance of the alternative hardware solutions by

running a "smaller" benchmark on a small cluster deployment.

In this chapter, we offer a novel framework, called ARIEL (Automated Resource Inference, Evaluation, and Learning), that aims to automate the system administrators' efforts on comparing different hardware choices for upgrading existing MapReduce clusters and sizing them for achieving targeted Service Level Objectives (SLOs). ARIEL consists of the following key components:

- *A set of parameterizable microbenchmarks* that is used to profile and measure different execution phases of MapReduce processing on a given platform. The parameters of these microbenchmarks affect the amount of data processed in the different phases by the map and reduce tasks. Our profiling technique measures the following five steps of the map task execution: *read, map, collect, spill,* and *merge*, and the following three phases in the reduce task processing: *shuffle, reduce,* and *write*. Intuitively, the execution of these phases depends on the performance of different hardware sub-parts comprising the system. For example, the *read* (*write*) phase depends on the I/O system read (write) performance, while the *merge* phase efficiency depends on both read and write throughput of the underlying I/O system. By executing these diverse benchmarks, we aim to create a useful training set that (implicitly) reflects the performance variations of these phases as a function of processed data and the hardware involved in the processing. Moreover, to enhance the microbenchmark generation, we offer an automated procedure for extracting parameter values that reflect properties of a given production workload.

- *A regression-based model* that establishes a relationship between performance of considered execution phases on the source system and their executions on the target destination platform. This model enables us to predict the MapReduce job performance on the new destination platform (without executing it). This goal is achieved by collecting the profile of a given job on the current platform (old hardware) and by applying the derived model to predict the phase durations on the new platform. Then, by combining the execution phases together, we can produce the job profile at the level of map and reduce tasks. Finally, these projected job profiles are used as the input to the analytic per-

formance models [30] and/or simulation tools [35] to accomplish the SLO-driven sizing of the future MapReduce cluster.

While the main scenario that we consider is the upgrade of the existing MapReduce cluster and sizing the future cluster, the proposed approach and technique can be applied to solve a broader set of related problems:

1. *Choosing hardware for a new MapReduce cluster.* There are many situations when the users are initiating the adoption of *Big Data* technologies. There might be a variety of hardware offerings for the future MapReduce cluster. The proposed approach offers a better understanding of comparative performance of the proposed hardware options and helps in choosing the appropriate entry point by assessing the additional cost/performance ratio.

2. *Comparing configuration options for a MapReduce cluster.* There is a variety of different configuration choices at the system level and for setting the Hadoop parameters. While there are many "rules of thumb" for navigating some of these choices, the proposed framework provides a general way of comprehensive comparison and understanding the impact of such choices. For example, one might evaluate the impact of additional hard disks on the performance of different phases in MapReduce processing pipeline. Intuitively, the performance of reading, writing, and merging phases might get improved, and the derived model will reflect the benefits of this change.

3. *Evaluating compute and storage offerings in the cloud for a MapReduce cluster deployment.* The proposed framework offers a unified approach for a detailed performance comparison of MapReduce processing phases not only of available options of the same cloud provider but across different cloud providers. The derived models help to predict the performance of given MapReduce applications on the target destination platforms and enable effective cluster sizing via simulation and replay of production workloads on the future Hadoop cluster. This helps in more accurate performance/cost comparison of different cloud options.

Our approach aims to eliminate error-prone manual processes and presents a fully automated solution. We validate our approach and its effectiveness

using a set of twelve realistic applications executed on three different hardware platforms. We justify the selection of profiling phases and the benchmark generation process by presenting the impact of these choices on the accuracy of the derived model. Our evaluation shows that the automated model generation procedure effectively characterizes different execution steps of MapReduce processing on three diverse hardware platforms. The predicted completion times of eleven applications (out of twelve) are within 10% of the measured completion times of the applications executed on the new platform.

The rest of this chapter is organized as follows. Section 6.2 discusses the problem definition. Section 6.3 presents our profiling approach, describes microbenchmarks, and explains objectives for their selection. Section 6.4 introduces the automated model generation for performance comparison of MapReduce processing on different hardware platforms. Section 6.5 evaluates the effectiveness of our model and benchmarks. We discuss the approach limitations and modeling challenges in Section 6.6. Section 6.7 concludes with a summary and future work directions.

6.2 Problem Definition and Our Approach

Migrating an existing MapReduce cluster and its applications to a new hardware is a challenging task with many performance questions to answer prior to the event. First of all, given a set of different hardware choices for a new MapReduce cluster, the system administrators need to evaluate and compare the performance of the upgrade candidates. However, comparing the performance of MapReduce clusters comprised of new hardware by inspecting the specifications of the underlying hardware parts is difficult even for a very experienced system administrator. Moreover, the intricate interaction of new generation processors, memory, and disks, combined with the complexity of the Hadoop execution model and layers of additional software such as HDFS (Hadoop Distributed File System) and JVM, make it difficult to predict the cluster performance by assessing the performance of underlying components with some traditional tests or benchmarks, e.g., by using the *dd* utility for evaluating the *read* and *write* throughput of the I/O sub-system, etc.

In this work, we discuss a general way of benchmarking and comparing the performance of MapReduce processing pipelines of different Hadoop clusters.

One of the end goals of this exercise is to be able to predict the performance of existing production jobs on the future, upgraded cluster. The performance profile of a MapReduce job can be characterized using durations (execution times) of its map and reduce tasks[1]. The overall completion time of the job depends on the cluster size and the amount of resources allocated to the job over time, i.e., the number of map and reduce slots assigned to the job's map and reduce tasks for processing. Typically, the required cluster size and related workload management decisions are evaluated via simulations, i.e., by replaying the job traces in the available MapReduce simulators, such as Mumak [31] or SimMR[35].

Therefore, given a job execution trace collected on the original, *old* Hadoop cluster, our goal is to produce its *scaled* version that reflects the job execution on the *new*, upgraded Hadoop cluster as shown in Figure 6.1.

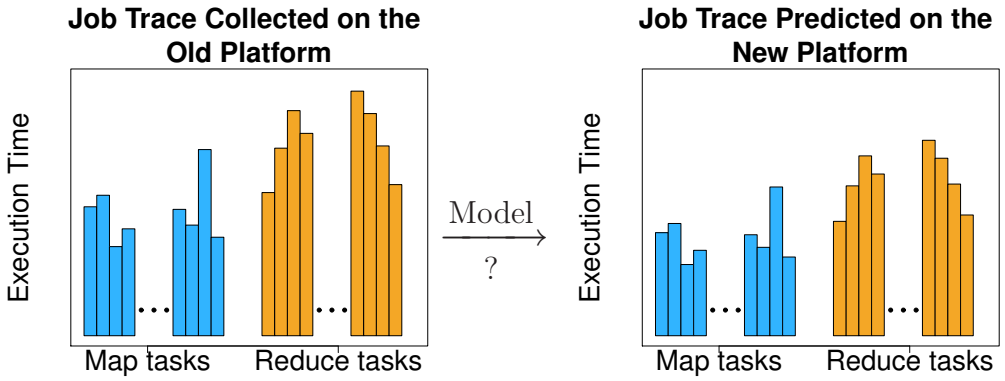


Figure 6.1: How to build a model from the old to the new platform?

Given such a trace transformation, we are able to predict the performance of production jobs on the new hardware and solve the sizing problem of a new cluster. Moreover, by having a collection of job traces that represent the production job executions on the new platform, many related workload management and job scheduling questions can be answered via pro-active simulations and informed decision making. To accomplish this goal, we design the following key components:

[1]The corresponding job trace can be extracted from the Hadoop job tracker logs using tools such as Rumen [32].

- *A set of parameterizable, synthetic microbenchmarks* to profile different execution phases of MapReduce processing on a given platform. These synthetic benchmarks process randomly generated data, and therefore, it is very easy to *scale up* and *down* the amount of data processed by the system as well as to define the amount of data flowing through different parts of the MapReduce processing pipeline. Our goal is to predict the performance of map and reduce tasks on the new platform. We decompose each map (reduce) task into a sequence of well-understood data processing steps or phases. The sum of the phase execution times defines the durations of map (reduce) tasks.

  The phase performance depends on the amount of data processed by it as well as the hardware efficiency of the underlying sub-systems that are involved in this phase. Note, that an attempt to derive an average or maximum throughput measurements for characterizing the performance of these phases might lead to very inaccurate prediction results. The reason is that the phase throughput does depend on the amount of data processed by the step, e.g., the *read* throughput is a function of the amount of read data ("small" reads have lower throughput compared to "large" reads). So, there is *no single value* that may characterize the phase performance. Therefore, a different modeling approach is needed and it is defined by the second component of our approach.

- *An automated model generation system* that establishes a relationship between execution of MapReduce processing phases on the source system and their executions on the target destination platform. This relationship is derived by running a set of microbenchmarks on each platform and building a model that relates the phase execution times of the same map and reduce tasks on both platforms as shown in Figure 6.2.

  Although it is created using data from synthetic benchmarks, the result is a general model which can be applied to any MapReduce job trace collected on the old platform to produce a *transformed job trace* that predicts the performance of this job on the new platform.

In this work, we do not aim to tune Hadoop parameters with ARIEL. We assume that the original (old) cluster parameters were already tuned to opti-

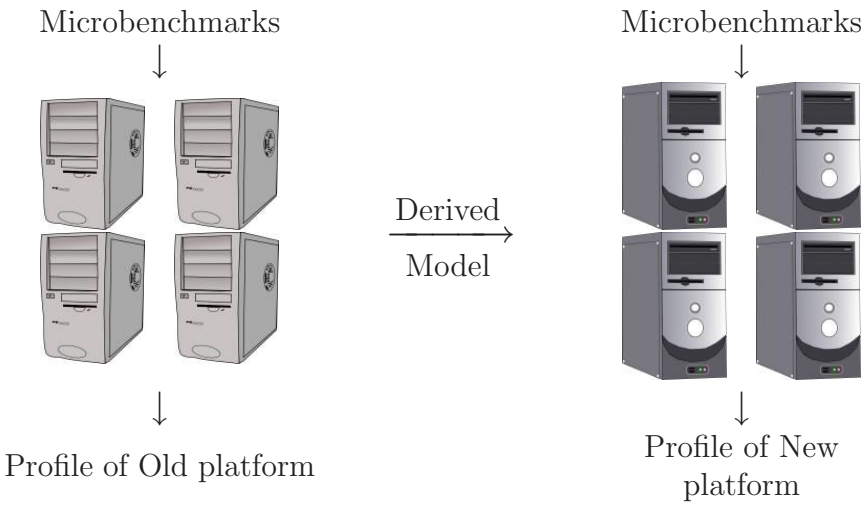


Figure 6.2: Automated model generation using benchmarking of both platforms.

mize the performance of production workloads. For benchmarking we deploy both Hadoop clusters with similar configuration parameters. A new platform might have a different number of cores per server. We apply an additional rule that Hadoop cluster on a new platform is configured with the same number of map and reduce slots per core as on the old one. Platform upgrades introduce a more powerful hardware, e.g., a higher amount of memory per core. While the performed experiments in the paper are based on slots configured with the same amount of memory on both platforms, the proposed approach does not require this. The slots (JVMs) on the new hardware might be allocated a larger amount of memory (if available).

The use of synthetic benchmarks ease the task of evaluating the cloud computing options available for deployment of a Hadoop cluster. Under the synthetic benchmarks' scenario, the user does not need to copy any real data or applications for performing an initial evaluation of available computing choices in the cloud.

6.3 Platform Profiling with Benchmarks

In this section, we describe the collection of microbenchmarks that are selected to profile and measure different execution phases of MapReduce pro-

cessing on a given platform. In order to determine a general relationship (a model) between the MapReduce processing pipelines of two different Hadoop clusters, we first accumulate the corresponding platform profiles by executing a specially selected set of microbenchmarks on the small deployments that represent both clusters.

6.3.1 Microbenchmark Suite

Our intent is to create a framework with a set of parameterizable, synthetic microbenchmarks to ease the way of creating a diverse variety of data processing patterns found in the production MapReduce workloads. These microbenchmarks enable us to measure the performance of different hardware platforms across a broad range of workload variations.

ARIEL supports microbenchmark generation through the following parameters:

1. *Input data size* (M_{inp}): This parameter controls the input read by each map task. We designed our microbenchmark such that each map task reads the entire file content as input. Hence, the input size is not limited to the HDFS block size (64MB or 128MB) and can be of arbitrary size. This parameter directly affects the *Read* phase duration.

2. *Map computation* (M_{comp}): We model the CPU computation performed by the user-defined map function by a simple loop which calculates the n^{th} Fibonacci number indicated by this parameter.

3. *Map selectivity* (M_{sel}): It is defined as the ratio of the map output to the map input. This parameter controls the amount of data produced as the output of the map function, and therefore it directly affects the *Collect*, *Spill* and *Merge* phase durations.

4. *Reduce computation* (R_{comp}): Similar to map computation, this parameter controls the computation performed by the reduce function by computing the n^{th} Fibonacci number.

5. *Reduce selectivity* (R_{sel}): It is defined as the ratio of the reduce output to the reduce input. This parameter specifies the amount of output

data written back to HDFS, and therefore it affects the *Write* phase duration.

Thus, the microbenchmark is parametrized as

$$B = (M_{inp}, M_{comp}, M_{sel}, R_{comp}, R_{sel}).$$

The user supplies a benchmark specification which consists of a list of values for each of the parameters as shown in Figure 6.3. Each benchmark consists

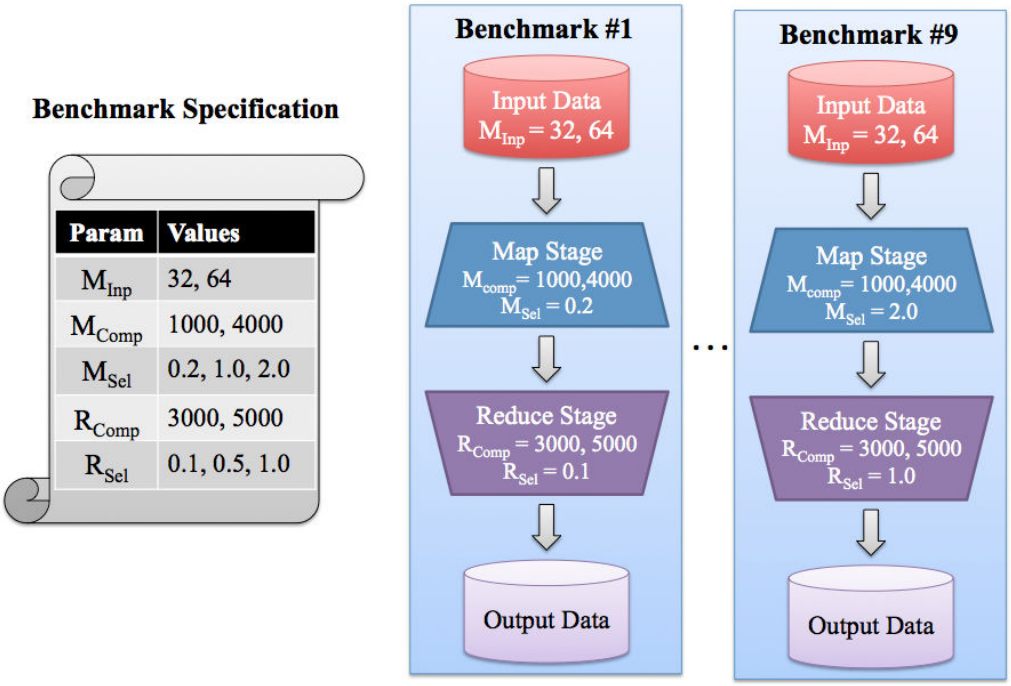


Figure 6.3: Microbenchmarking methodology.

of a specified (fixed) number of map and reduce tasks. For example, we generate benchmarks with 20 map and 20 reduce tasks each for execution in our small cluster deployments with 5 worker nodes (see setup details in Section 6.5). Increasing number of tasks might improve the accuracy of constructed model, but will require a longer benchmarking time.

The benchmarking engine then generates input data consisting of 100 byte key/value pairs using *TeraGen* [38], a Hadoop utility for generating synthetic data. The size of the input data for each task is selected in a round robin fashion from the list of input data size values in the benchmark specification. This data is used by each microbenchmark. For every value of M_{sel} and R_{sel},

a new microbenchmark is created and executed. Thus, there are $M_{sel} \times R_{sel}$ number of microbenchmarks executed in total. Within each MapReduce microbenchmark, the map tasks round robin through the list of M_{comp} values and the reduce tasks round robin through the list of R_{comp} values as shown in Figure 6.3.

The user can quickly create a microbenchmark suite that covers useful and diverse ranges across all five parameters. This suite can be used for comparison of hardware options when no additional description of specific workloads of interest is provided or during a preliminary step of high-level hardware assessment for narrowing down the multiple choices.

When a set of critical applications and production jobs for migration is known, we offer an automated procedure for extracting *customized* parameter ranges that reflect properties of a given workload. One can enhance the microbenchmark generation and coverage by using a *customized* list of extracted parameters. This helps to enrich the collection of training data for generating a more accurate prediction model. In order to cover the parameter space of a given production workload, we first profile this workload to extract the parameters of each application in the set. Subsequently, these profiles are combined to build the CDF (Cumulative Distribution Function) of parameter values exercised by these applications. Figure 6.4 shows the CDFs of five parameters (*input data sizes*, *map* and *reduce selectivities*, *map* and *reduce computations*) for a set of 12 applications used in the performance evaluation study in Section 6.5.

From the CDF plots, we can determine the appropriate values for the microbenchmark parameters by using a clustering algorithm like k-means. Red rhombuses in Figures 6.4 (a)-(e) show the outcome of the clustering algorithm for $k = 4$. By choosing a smaller or larger number k of clusters in the k-means algorithm, we can influence the number of parameter values (and the number of benchmarks) used in the microbenchmark suite to control the overall benchmarking time. Typically, a benchmark specification that better mimics the parameters of the production workload helps to improve the accuracy of the constructed prediction model.

Figure 6.4 (f) shows the relationship between the Fibonacci number n (shown on x-axis) and its computation time (shown on y-axis). This relationship is used for defining durations of map and reduce phases whose CDFs are shown in Figures 6.4 (d)-(e) respectively. We should point out that the

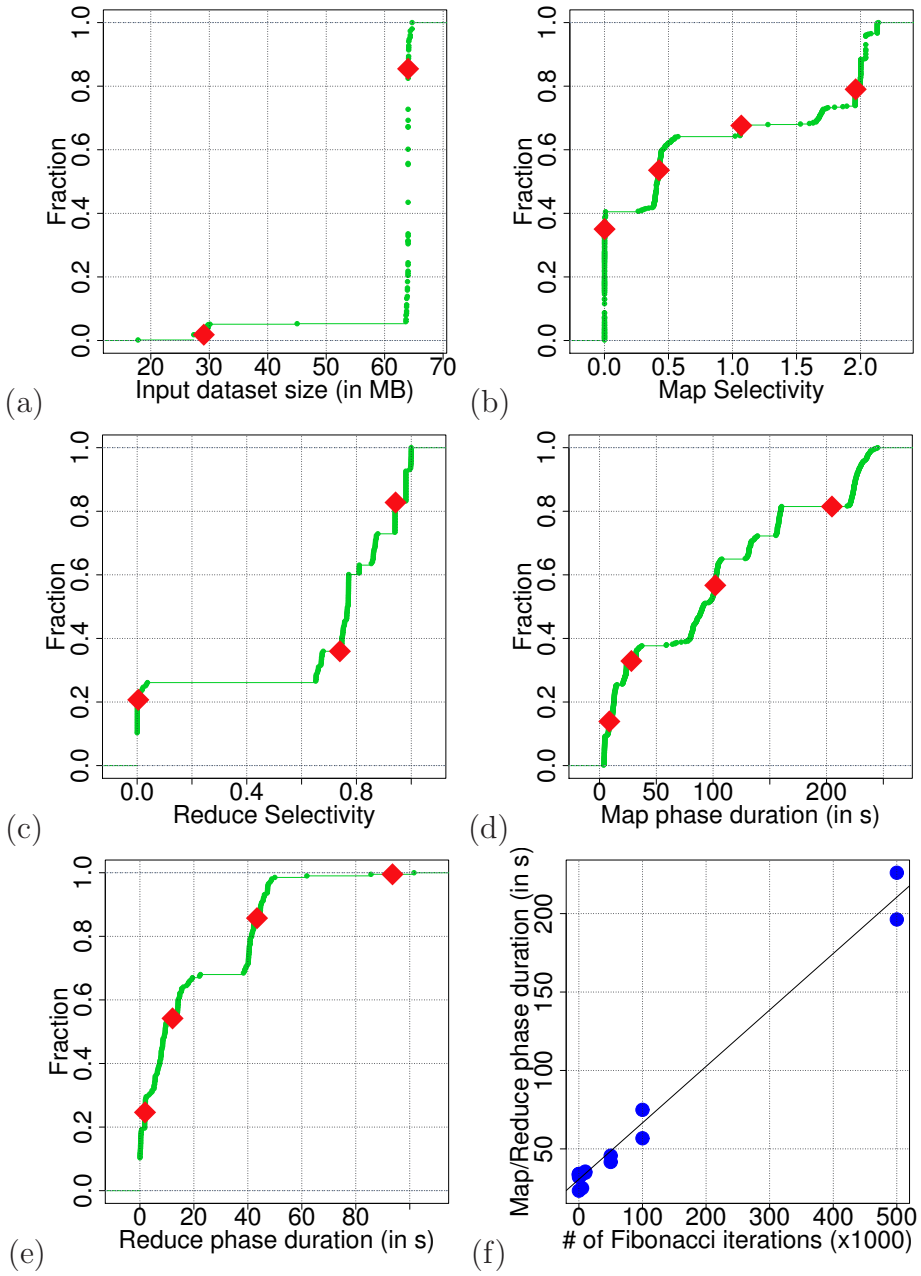


Figure 6.4: Application parameters

proposed approach of defining map and reduce computation phases via Fibonacci numbers computation is clearly simplistic because custom map and reduce functions might specify complex computation routines. We defer the discussion of challenges related to benchmarking and modeling of map and reduce computation phases to Section 6.6.

6.3.2 Profiling Execution Phases of MapReduce Processing Pipeline

The execution of a map (reduce) task is comprised of a specific, well-defined sequence of processing phases. Intuitively, the execution of a particular phase depends on the amount of data flowing through this phase and the performance of different hardware sub-parts involved at this step. After the cluster upgrade, when this processing pipeline is executed on a different platform, the performance of underlying hardware might impact the performance of these execution phases in a different way, and therefore, the scaling factors of these phases with respect to the old platform might be different as well. In order to construct an accurate predictive model, we need to profile the durations of these phases during the task execution, and then determine their individual scaling functions from the collected measurements.

We measure five phases of the map task execution and three phases of the reduce task execution. Map task processing consists of the following phases:

1. *Read* – it measures the time taken to read the map inputs from the distributed file system. A map task typically reads a block (64MB or 128MB). However, map tasks of many applications read entire files or compressed files of varying sizes. The *read* duration is primarily a function of the disk read throughput.

2. *Map* – it measures the time taken for executing the user supplied map function on the input key-value pair. It depends on the CPU performance.

3. *Collect* – it measures the time taken to buffer map phase outputs into memory. It is a function of the memory bandwidth of the system.

4. *Spill* – it measures the time taken to (locally) sort the intermediate data and partition for the different reduce tasks, applying the combiner if available, and writing the intermediate data to local disk. Multiple system components impact this phase performance.

5. *Merge* – it measures the time taken to merge the different spill files into a single spill file for each reduce task. *Merge* phase depends on the disk read and write throughput.

Reduce task processing consists of the following phases:

1. *Shuffle* – it measures the time taken to transfer intermediate data from map tasks to the reduce tasks and merge-sorting them together. We combine the shuffle and sort phases because in the Hadoop implementation, these two sub-phases are interleaved. The shuffle duration primarily depends on the network shuffle performance and disk read and write throughput.

2. *Reduce* – it measures the time taken to apply the user supplied reduce function on the input key and all the values corresponding to it. Like the *map* phase, it depends on the CPU performance.

3. *Write* – it measures the amount of time taken to write the reduce output to the distributed file system. This operation depends on the disk write (and possibly network) throughput.

In the next section, we describe how these measurements are utilized for building the platform profiles.

6.3.3 Platform Profiles

We generate *platform profiles* by running a set of our microbenchmarks on the Hadoop clusters being compared. After the execution of each microbenchmark, we gather durations of the execution phases of all processed map and reduce tasks. A set of these measurements defines the *platform profile* that is used as the training data for the model. We collect durations of *eight* execution phases that reflect essential steps in processing of map and reduce tasks on the given platform:

- *Map task processing:* in the collected platform profiles, we denote the phase duration measurements for *read, map, collect, spill,* and *merge* as D_1, D_2, D_3, D_4, and D_5 respectively.

- *Reduce task processing:* in the collected platform profiles, we denote phase duration measurements for *shuffle, reduce,* and *write* as D_6, D_7, and D_8 respectively.

Tables 6.1, 6.2 show two fragments of a collected platform profile as a result of the executed benchmarking set.

Bench-mark *ID*	Map Task *ID*	*Read* *msec* D_1	*Map* *msec* D_2	*Collect* *msec* D_3	*Spill* *msec* D_4	*Merge* *msec* D_5
1	1	1010	220	610	5310	10710
1	2	1120	310	750	5940	11650
...	...	...	...	...	...	...

Table 6.1: A fragment of a collected platform profile: map task processing.

Bench-mark *ID*	Reduce Task *ID*	*Shuffle* *msec* D_6	*Reduce* *msec* D_7	*Write* *msec* D_8
1	1	10110	330	2010
1	2	9020	410	1850
...	...	...	...	...

Table 6.2: A fragment of a collected platform profile: reduce task processing.

6.3.4 Profiling Overhead

Job traces must be gathered from the production MapReduce applications running on the source platform. Hence, it is important for our monitoring environment to be lightweight. To obtain phase durations, we design a profiling tool inspired by Starfish [39] based on *BTrace* – a dynamic instrumentation tool for Java [40]. BTrace intercepts class byte codes at run-time based on event-condition-action rules and injects byte codes for the associated actions. Dynamic instrumentation has the appealing property that it has a zero overhead when monitoring is turned off. Also, no code modifications to the deployed production framework and applications are needed. We instrument selected Java classes and functions internal to Hadoop using BTrace and measure the time taken for executing different phases.

We also explore an alternative profiling implementation to reduce the profiling overhead. Currently, Hadoop includes several counters such as number of bytes read and written. These counters are sent by the worker nodes to the master periodically with each heartbeat. We modified the Hadoop code by adding counters that measure the durations of *eight phases* to the existing counter reporting mechanism. We evaluate the runtime overhead of different profiling techniques in Section 6.5.7.

Apart from the phases described in Section 6.5.4, each task has a constant overhead for setting and cleaning up. Moreover, the profiling process itself

incurs some additional overhead per task (typically, it is a function of the task duration). We account for these overheads separately for each task.

6.4 Model Generation

This section describes how to create a model $M_{src \to dst}^{ARIEL}$ which characterizes the relationship between MapReduce job executions on two different Hadoop clusters, denoted here as *src* and *dst* clusters. To accomplish this goal, we first, find the relationships between durations of different execution phases on the given Hadoop clusters, i.e., we build *eight* submodels $M_1, M_2, ..., M_7, M_8$ that define the relationships for *read, map, collect, spill, merge, shuffle, reduce,* and *write* respectively on two given Hadoop clusters. To build these submodels, we use the platform profiles gathered by executing a set of microbenchmarks on the original Hadoop cluster and the target destination platform (see Tables 6.1, 6.2.)

Below, we explain how to build a submodel M_i, where $1 \leq i \leq 8$. By using values from the collected platform profiles, we form a set of equations which express duration of the specific execution phase on the destination platform *dst* as a linear function of the same execution phase on the source platform *src*. Note that the right and left sides of equations below relate the phase duration of the same task (map or reduce) and of the same microbenchmark on two different platforms (by using the task and benchmark *ID*s):

$$\begin{aligned} D_{i,dst}^{1,1} &= A_i + B_i * D_{i,src}^{1,1} \\ D_{i,dst}^{1,2} &= A_i + B_i * D_{i,src}^{1,2} \\ .... & \quad .... \\ D_{i,dst}^{2,1} &= A_i + B_i * D_{i,src}^{2,1} \\ .... & \quad .... \end{aligned} \tag{6.1}$$

where $D_{i,src}^{m,n}$ and $D_{i,dst}^{m,n}$ are the values of metric D_i collected on the source and destination platforms for the task with $ID = n$ during the execution of microbenchmark with $ID = m$ respectively.

To solve for (A_i, B_i), one can choose a regression method from a variety of known methods in the literature (a popular method for solving such a set of equations is a non-negative Least Squares Regression).

Let $(\hat{A}_i, \hat{B}_i)$ denote a solution for the equation set (1). Then $M_i = (\hat{A}_i, \hat{B}_i)$ is the submodel that describes the relationship between the duration of execution phase i on the source and destination platforms. The entire model $M^{ARIEL}_{src \to dst} = (M_1, M_2, ..., M_7, M_8)$.

Our training dataset is gathered by an automated benchmark system which runs identical benchmarks on both the source and destination platforms. The non-determinism in MapReduce processing, some unexpected anomalous or background processes can skew the measurements, leading to outliers or incorrect data points. With ordinary least squares regression, even a few bad outliers can significantly impact the model accuracy, because it is based on minimizing the overall absolute error across multiple equations in the set.

To decrease the impact of occasional bad measurements and to improve the overall model accuracy, we employ iteratively re-weighted least squares [41]. This technique is from the Robust Regression family of methods designed to lessen the impact of outliers.

6.5 Evaluation

In this section, we assess the validity of the proposed approach and justify the design choices in ARIEL: Why do we need to distinguish different *phases* for application profiling? What is the impact of varying the different *parameters* for generating the *microbenchmarks*? We evaluate the accuracy of our models and the effectiveness of the overall approach using a suite of twelve applications and three hardware platforms.

6.5.1 Platforms

We consider the following hardware platforms for a Hadoop cluster (we name them for simplicity):

1. *old:* HP DL145 G3 machines with four AMD 2.39 GHz cores, 8 GB RAM, two 160 GB 7.2K rpm SATA hard disks, and interconnected with gigabit Ethernet.

2. *new1:* HP DL380 G4 machines with eight Intel Xeon 2.8 GHz cores, 6 GB RAM, four 300 GB 10K rpm SATA drives, and interconnected

with gigabit Ethernet.

3. *new2:* HP DL160 G6 machines with eight Intel Xeon 2.66 GHz cores, 16 GB RAM, two 1 TB 7.2K rpm SATA drives, and interconnected with gigabit Ethernet.

6.5.2 Applications

To evaluate the accuracy of ARIEL we use a set of twelve applications made available by Tarazu project [42]:

1. *Sort* sorts randomly generated 100-byte tuples. The sorting occurs in MapReduce framework's in-built sort while map and reduce are identity functions.

2. *WordCount* counts all the unique words in a set of documents.

3. *Grep* searches for an input string in a set of documents.

4. *InvertedIndex* takes a list of documents as input and generates word-to-document indexing.

5. *RankedInvertedIndex* takes lists of words and their frequencies per file as input, and generates lists of files containing the given words in decreasing order of frequency.

6. *TermVector* determines the most frequent words on a host (above a specified cut-off) to aid analysis of the host's relevance to a search.

7. *SequenceCount* generates a count of all unique sets of three consecutive words per document in the input data.

8. *SelfJoin* is similar to the candidate generation part of the Apriori data mining algorithm. It generates association among k+1 fields given the set of k-field associations and uses synthetically generated data as input.

9. *AdjacencyList* is useful in web indexing to generate adjacency and reverse adjacency lists of nodes of a graph for use by PageRank-like algorithms. It uses synthetically generated web graph based on a Zipfian distribution.

Application	*Input data type*	*Input (GB) small*	*#Map, Reduce tasks*	*Input (GB) large*	*#Map, Reduce tasks*
Sort	Synthetic	2.8	44, 20	31	495, 240
WordCount	Wikipedia	2.8	44, 20	50	788, 240
Grep	Wikipedia	2.8	44, 1	50	788, 1
InvIndex	Wikipedia	2.8	44, 20	50	788, 240
RankInvIndex	Wikipedia	2.5	40, 20	47	745, 240
TermVector	Wikipedia	2.8	44, 20	50	788, 240
SeqCount	Wikipedia	2.8	44, 20	50	788, 240
SelfJoin	Synthetic	2.1	32, 20	28	448, 240
AdjList	Synthetic	2.4	44, 20	28	508, 240
HistMovies	Netflix	3.5	56, 1	27	428, 1
HistRatings	Netflix	3.5	56, 1	27	428, 1
KMeans	Netflix	3.5	56, 16	27	428, 16

Table 6.3: Application characteristics

10. *HistogramMovies* generates a histogram of the number of movies with different average ratings (from 1 to 5).

11. *HistogramRatings* generates a histogram of all user ratings (ranging from 1 to 5).

12. *KMeans* clusters movies into k clusters using the cosine-vector similarity and recomputes the new centroids.

Applications 1, 8, and 9 process synthetically generated data, applications 2 to 7 use the Wikipedia articles dataset as input, while applications 10 to 13 use the Netflix movie ratings dataset.

Table 6.3 summarizes these 12 applications. We will present results of running these applications with: *i) small input datasets* defined by parameters shown in columns 3-4, and *ii) large input datasets* defined by parameters shown in columns 5-6 respectively.

6.5.3 Motivating Scenario

Our performance study is motivated by the following typical scenario of a system administrator managing a Hadoop cluster. Our administrator has a 60-node Hadoop cluster based on the *old* hardware (see details in Section 6.5.1).

He has a choice of two new hardware platforms: *new1* and *new2*, and needs to compare these choices. It is not obvious which platform might be a better performing one: from the high-level specifications the *new1* platform has a slightly faster server and faster disks compared to the *new2* platform[2]. Once the new upgrade platform is chosen, the administrator needs to project the expected performance of the 12 applications (the set of critical production jobs) on the new platform for cluster sizing and workload management goals.

The *old* Hadoop cluster is configured with one map and one reduce slot per core, and 700 MB of RAM per slot. For benchmarking, we create two small Hadoop clusters with 5 worker nodes (plus one node for Hadoop management) based on *new1* and *new2* platforms. These clusters are configured similarly, i.e., one map and one reduce slots per core, and 700 MB of RAM per slot.

First, we execute the set of our microbenchmarks on the small 5-nodes clusters of *new1* and *new2* platforms. Then we create the corresponding platform profiles and build a model $M^{ARIEL}_{new1->new2}$. We use this model for comparing the performance of the two platforms. Figure 6.5 shows the relationship for executing eight phases on the *new1* and *new2* platforms. Each graph has a collection of dots that represent phase measurements of map (reduce) tasks on two given platforms, i.e., for each dot, the X-axes value represents the phase duration on the *new1* platform while the corresponding Y-axes value represents the phase duration on the *new2* platform. The red line on the graph shows the linear regression solution that serves as a model for this phase. As we can see (visually) the linear regression provides a good solution for all the phases. Here is the derived model: $M^{ARIEL}_{new1\rightarrow new2} = (M_1, M_2, ..., M_7, M_8) = (0.40, 0.44, 0.43, 0.39, 0.31, 0.68, 0.37, 1.33)$. For simplicity we only show the slope values B_i and omit intercept values A_i (because the intercepts are close to 0).

The model means that if *read* takes 1000 msec on the *new1* platform then *read* execution on the *new2* platform takes on average 400 msec (according to the submodel M_1 value). However, if *write* takes 1000 msec on *new1* platform then *write* execution on the *new2* platform takes on average 1330 msec (according to the submodel M_8 value). This is an interesting observation

[2]We ask the reader do not concentrate on a difference of RAM and disk capacities of the given hardware here. Our goal is to offer the approach for comparing the platforms' performance. RAM and disk capacities can be always easily compared and matched to satisfy the customer needs. One may assume, that RAM and disk capacities in the considered choices are the same.

that stresses the importance of understanding the workload of interest. In our case, *new2* platform outperforms *new1* platform in the overall map and reduce tasks performance and in spite of slower write operation represents a more efficient platform.

6.5.4 Importance of Modeling Execution Phases

While we can see that different phases on these platforms have different scaling functions, the question still remains open whether one can try to build a simpler model, for example, based on profiling map and task durations instead of creating a detailed 8-phase based profile. To answer this question we collect the platform profiles based on durations of map and reduce tasks, and use linear regression to create the model $M^{task}_{new1 \rightarrow new2}$. Figures 6.6 (a)-(b) show the durations of map and reduce tasks on *new1* and *new2* platforms after executing the set of our microbenchmarks and the model (red line) derived using the linear regression approach. From these figures we can see that the linear regression model does not provide a good fit to the underlying training data.

In order to formally compare the accuracy of generated models $M^{ARIEL}_{new1 \rightarrow new2}$ and $M^{task}_{new1 \rightarrow new2}$ we compute per task prediction error. For each task i and its measured duration d_i in our platform profile of the *new1* hardware we compute the task predicted duration d_i^{pred} on the *new2* platform using the derived model. Then we compare the predicted value against the measured duration d_i^{measrd} of the same task i on the *new2* platform. The relative error is defined as follows:

$$error_i = \frac{|d_i^{measrd} - d_i^{pred}|}{d_i^{measrd}}$$

We compute the relative error for all the tasks in the platform profile. Figures 6.6 (c)-(d) show the CDF of relative errors for map and reduce tasks using two models: *phase-based* $M^{ARIEL}_{new1 \rightarrow new2}$ and *task-based* $M^{task}_{new1 \rightarrow new2}$. We observe that the accuracy of the *task-based* model is much worse compared to the accuracy of the *phase-based* model. Under the *task-based* model 44% of map tasks and 26% of reduce tasks have the prediction error higher than 100%. Therefore, if we derive the individual scaling submodels for execution phases of the MapReduce processing pipeline then the quality of the overall prediction model is significantly higher. It will help to accurately project the

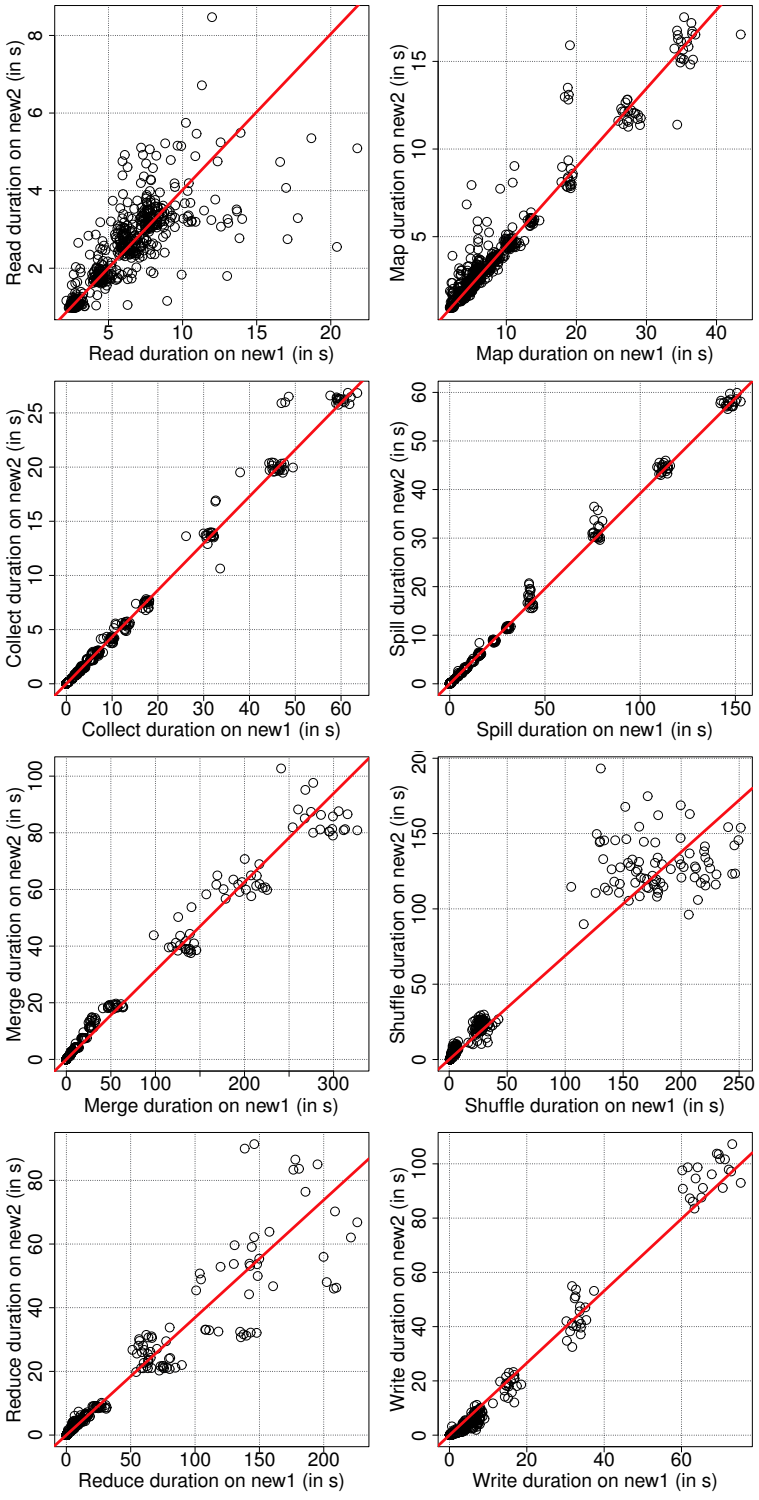


Figure 6.5: Different phase durations on *new1* and *new2* platforms.

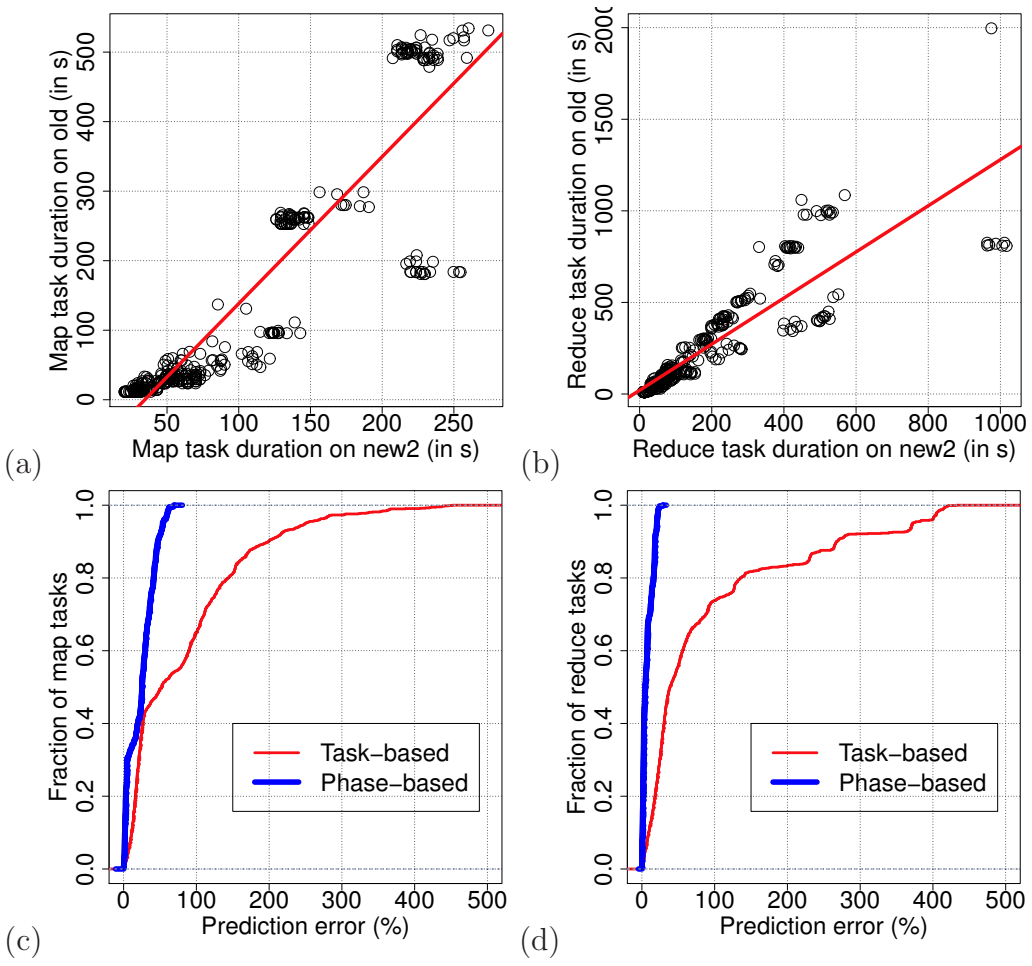


Figure 6.6: Do we need phases for application profiling? (a)-(b) platform profiles based on map and reduce task durations, (c)-(d) CDF of relative errors for two models: *phase-based* $M^{ARIEL}_{new1 \rightarrow new2}$ and *task-based* $M^{task}_{new1 \rightarrow new2}$.

execution of a MapReduce application from one platform to a different one and reflect the specifics of this job execution.

Figure 6.7 shows the job traces for WordCount application. Figures 6.7 (a)-(b) show the durations of map and reduce tasks measured on the *new1* platform, and Figures 6.7 (c)-(d) show the predicted durations of map and reduce tasks for their execution on the *new2* platform. The "appearance" of measured and predicted job traces closely resemble each other, while the predicted durations of map and reduce tasks have almost two times difference in absolute values. The proposed approach can capture the specifics of the application execution, e.g., due to existing skews in the popularity of the reduce keys as one can see in Figures 6.7 (b) and 6.7 (d).

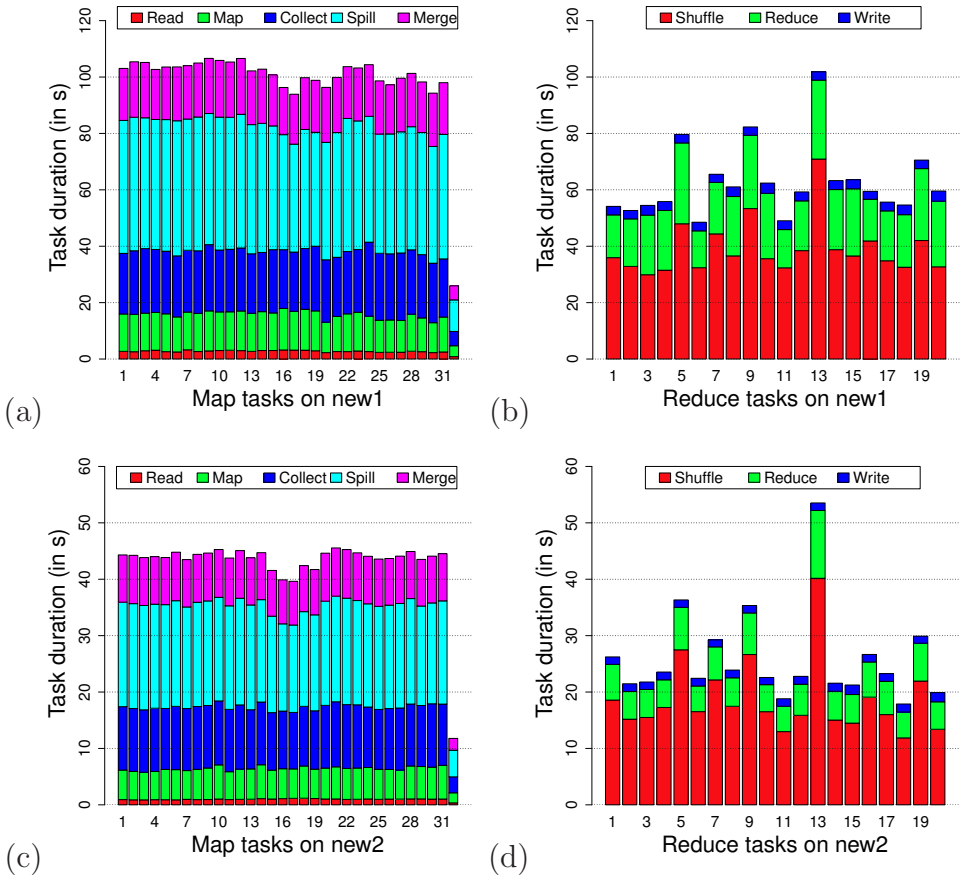


Figure 6.7: Job traces (map and reduce tasks durations) of WordCount application: (a)-(b) *new1* platform; and (c)-(d) *new2* platform.

Accurately scaled job traces on the new platform enable different modeling styles of the job completion time: analytic and simulation ones. Figure 6.8 presents the predicted and measured job completion times of 12 applications introduced in Section 6.5.2. In these experiments we use applications with *small input datasets* shown in Table 6.3. First, we run these applications on 5-node Hadoop cluster based on the *new1* platform. Then we apply the derived linear regression model $M^{ARIEL}_{new1 \rightarrow new2}$ to the collected job traces to transform them into the job traces on the *new2* platform. Finally, we predict completion times of these applications on the 5-node Hadoop cluster with *new2* hardware. For a prediction we have applied two different approaches:

- An *analytical model* ARIA [30] that utilizes the knowledge of the average and maximum durations of map and reduce tasks and provides a simple model for predicting the job completion time as a function

of allocated resources. Since we have the entire tasks distribution it is easy to compute the average and maximum task durations.

- A *MapReduce simulator SimMR* [35] that supports a job trace replay on a Hadoop cluster as a way to more accurately predict the job completion time (especially under the different Hadoop schedulers).

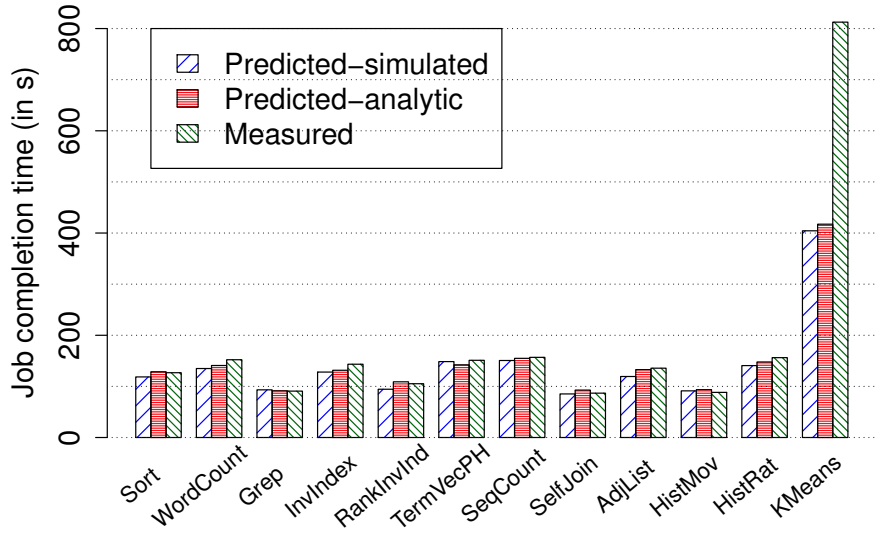


Figure 6.8: Predicted vs. measured completion times of 12 applications on 5-node cluster with *new2* hardware.

Figure 6.8 shows that errors between the predicted and job completion times are on average 6% (and within 10%) for 11 applications out of 12. The prediction error for *KMeans* is almost 50%. This application has complex, compute-intensive map and reduce functions that are difficult to predict accurately with a black-box approach. We offer an additional discussion on the causes of high prediction error in Section 6.6. However, the majority of MapReduce applications have relatively simple data manipulations performed by their map and reduce functions, and these applications can be accurately modeled by the "black-box" approach proposed in the paper.

Figure 6.9 presents a different perspective on the accuracy of the designed linear-regression model. It shows the CDF of relative errors in predicting map and reduce task durations of 12 applications (combined) on the *new2* platform. For 80% of the map and reduce tasks the relative error is less than

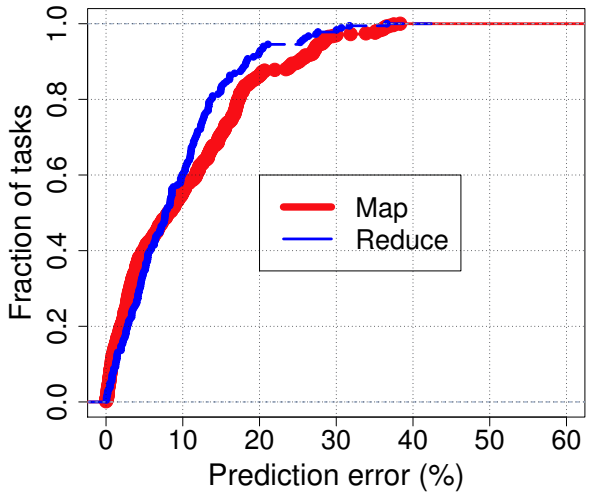


Figure 6.9: CDF of relative errors for predicting task durations of 12 applications with $M^{ARIEL}_{new1 \rightarrow new2}$ model.

18%. We observe that the overall job completion time prediction is quite accurate (less than 10% error for 11 applications) in spite of less accurate prediction of the task durations. Partially, this is because Figure 6.9 shows the absolute relative error. However, during multiple waves of task execution some of these positive and negative errors in predicted task durations cancel each other.

6.5.5 Benchmark Coverage

In these experiments, we examine the importance of the introduced parameters for the microbenchmark generation. We aim to answer the question whether one can choose a smaller set of parameters and execute a much smaller subset of microbenchmarks for building a good model. For example, how important is varying the map or reduce selectivities for creating the representative benchmarking set and generating an accurate model? To answer this question we also fix the input data size since having a diverse input size for different map tasks partially creates the effect of different map and reduce selectivities in the MapReduce jobs. We create two models $M^{fix\_map\_sel}_{new1 \rightarrow new2}$ and $M^{fix\_red\_sel}_{new1 \rightarrow new2}$ using two sets of microbenchmarks generated by the following specifications respectively:

- Fixed input dataset size per map task and *fixed map selectivity* while varying other parameters;
- Fixed input dataset size per map task and *fixed reduce selectivity* while varying other parameters.

We apply these models to predict map and reduce task durations of 12 applications (described in Section 6.5.2) on the *new1* platform. In order to formally compare the accuracy of these two models and the model $M^{ARIEL}_{new1 \rightarrow new2}$ that is generated using the set of all the benchmarks (i.e., while varying all the parameters) we compute the CDF of per task relative prediction errors (as defined in Section 6.5.4). Figure 6.10 summarizes the outcome of these experiments:

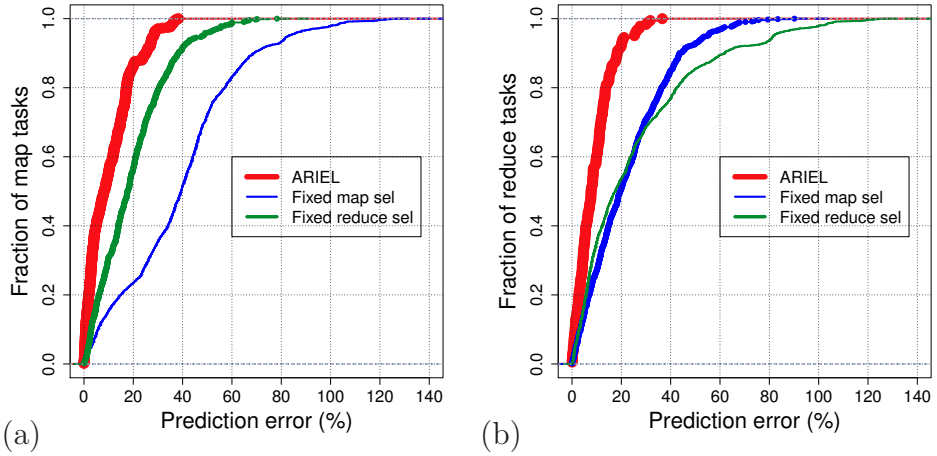


Figure 6.10: CDF of relative errors for predicting task durations of 12 applications with different models: $M^{ARIEL}_{new1 \rightarrow new2}$, $M^{fix\_map\_sel}_{new1 \rightarrow new2}$, and $M^{fix\_red\_sel}_{new1 \rightarrow new2}$.

Figure 6.10 (a) shows the CDF of errors for the three models when predicting the duration of map tasks. We observe that that the accuracy of the models generated by the benchmarks with fixed map or reduce selectivities is significantly worse compared to the accuracy of the model built using all the benchmarks (where we used varying set of values for defining map and reduce selectivities). The accuracy of prediction for reduce tasks shown in Figure 6.10 (b) presents a similar trend. For example, under the model obtained using benchmarks with *fixed map selectivity* 50% of map tasks and 15% of reduce tasks have the prediction error higher than 40%.

6.5.6 Model Accuracy

In this section, we follow the motivating scenario described in Section 6.5.3. After comparing two hardware platforms *new1* and *new2*, the system administrator selects a better performing *new2* platform. Now, the next task is to project the expected performance of 12 applications (the set of critical production jobs) on the new platform for cluster sizing and workload management goals. Under our approach we need to benchmark both platforms and then build a model using the platform profiles. Since we have already benchmarked the *new2* hardware and have built its platform profile we only need to execute the suite of microbenchmarks on the original *old* platform and extract its platform profile. Here is the derived model: $M^{ARIEL}_{old \rightarrow new2}$ $= (0.2, 0.17, 0.37, 0.6, 0.46, 0.86, 0.29, 0.32)$. For simplicity we only show the slope values B_i for each submodel and omit the intercept values A_i (since they are close to 0).

For example, it means, that if *read* takes 1000 msec on the *old* platform then the *read* phase execution on the *new2* platform takes on average only 200 msec. One can see that the execution phases are much more efficient of the new platform. The *shuffle* phase has somewhat similar durations on both platforms because of a similar networking infrastructure in both clusters.

To evaluate the model accuracy, we perform a set of experiments with 12 applications (that process *large* input datasets as shown in Table 6.3) on the large Hadoop clusters based on the *old* and *new2* platforms with 60 worker-nodes each. Figure 6.11 shows the CDF of relative errors in predicting map and reduce task durations of 12 applications (combined) on the *new2* platform. For 88% of the tasks the relative error is less than 20%.

Finally, Figure 6.12 shows that the error between predicted and measured job completion times are on average 6% (and within 10%) of the measured completion times for 11 applications out of 12. The prediction error for *KMeans* is still high, around 30% due to its complex map and reduce functions that are difficult to predict with a black-box approach. In Section 6.6 we discuss a way to assess a possible prediction inaccuracy due to map and reduce function executions.

For the remaining 11 applications, we observe that the model derived by benchmarking a *small* 5-node test cluster enables an accurate prediction of job completion times in the *larger* production cluster deployment.

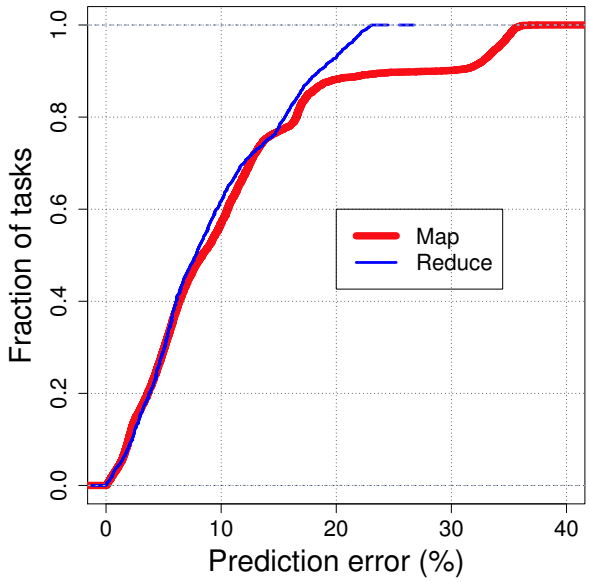


Figure 6.11: CDF of relative errors for predicting task durations of 12 applications processing *large* datasets with $M^{ARIEL}_{old \rightarrow new2}$ model on the 60-node *new2* cluster.

6.5.7 Profiling Overhead

In order to evaluate the profiling overhead incurred by different profiling techniques, we execute our set of twelve applications on the *new2* platform without any profiling, with counter-based profiling and with BTrace-based profiling. Figure 6.13 shows the the job completion time for the different profilers.

We observe that counter-based profiling adds an average of 8% (and up to 13%) overhead. BTrace-based profiling has an average of 10% (and up to 28%) overhead. The compute intensive KMeans application has the highest overhead, since the map and reduce functions compete with the profiler for CPU resources. However, both profilers have a modest overhead even when they are profiling each task execution.

6.6 Discussion

In this section, we discuss some limitations of the proposed approach and provide comments on how to improve the modeling accuracy with ARIEL.

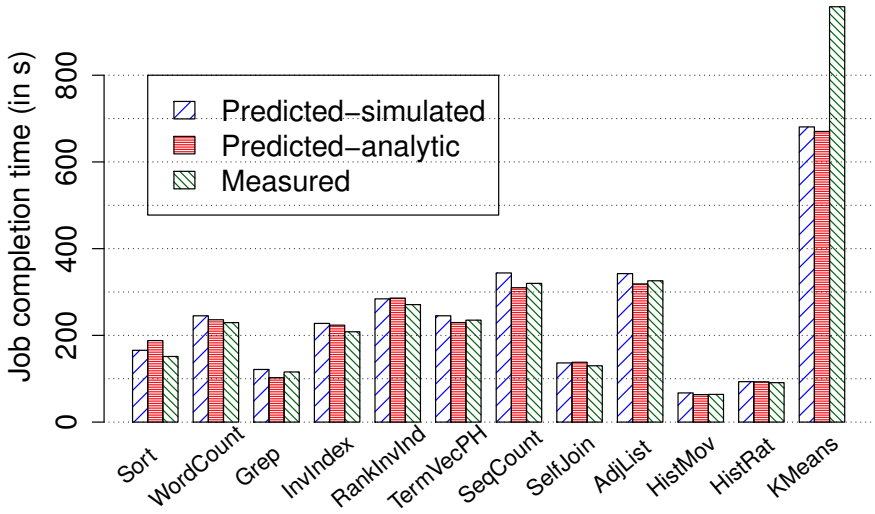


Figure 6.12: Predicted vs. measured completion times of 12 applications on 60-node cluster with *new2* hardware.

Modeling compute-intensive map and reduce functions: Some MapReduce programs, e.g., *KMeans*, involve complex, compute-intensive map/reduce functions. Figure 6.14 shows that 94% of the map task duration and 30% of the reduce task duration are due to *map* and *reduce* phase processing respectively.

Our simple approximation of computing power of different platforms with Fibonacci number iterations cannot provide an accurate scaling function for complex computations. In order to provide quick feedback to the user about possible prediction inaccuracy, we can perform a quick application analysis by inspecting the fraction of the map and reduce phase durations in the entire task duration.

First shuffle phase measurements and modeling: a shuffle phase for a "first wave" of reduce tasks overlaps with the map stage execution. We model the *first shuffle* separately by including the non-overlapping portion of the execution time (similar to a modeling technique designed in ARIA [30]) and building a separate linear model for this phase.

Scaling network resources: The set of twelve applications can be executed in 115 minutes using the FIFO scheduler on the 60-node *old* cluster. The following table shows the number of *new2* platform machines required

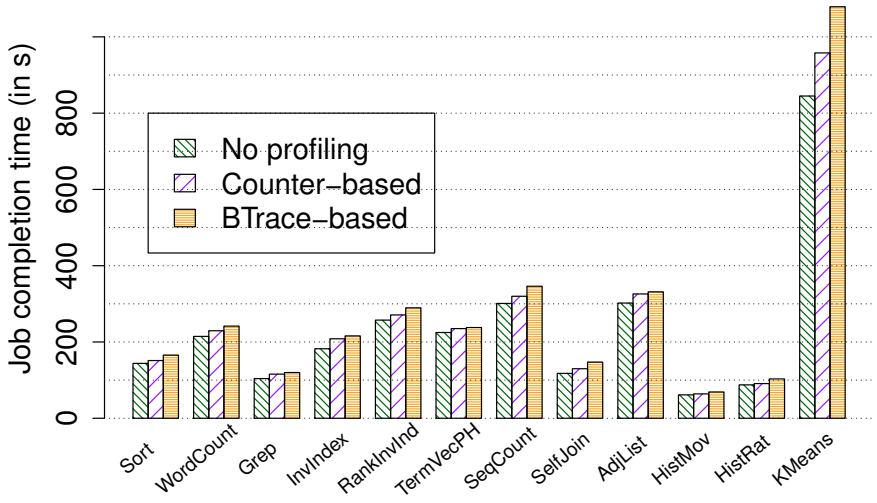


Figure 6.13: Job completion time with different profilers.

to satisfy a given SLO. These estimates are obtained with our MapReduce simulator SimMR [35]. It shows that to achieve the same completion time on a *new2* platform, we need 16-node cluster.

SLO (mins)	120	115	60	45
# machines	15	16	30	64

Table 6.4: Resource allocation estimates for different SLOs.

However, as a larger number of machines have to process the same amount of data in a shorter time, the network should be capable of supporting a higher bisection bandwidth as compared to the *old* platform. Analysis of required network resources can be performed using simulators like ns-2. In the future work, we plan to design network microbenchmarks that transfer varying amounts of intermediate data from memory of the machines processing map tasks to memory of the machines processing the reduce tasks (without any disk accesses) and thus accurately measure the shuffle performance.

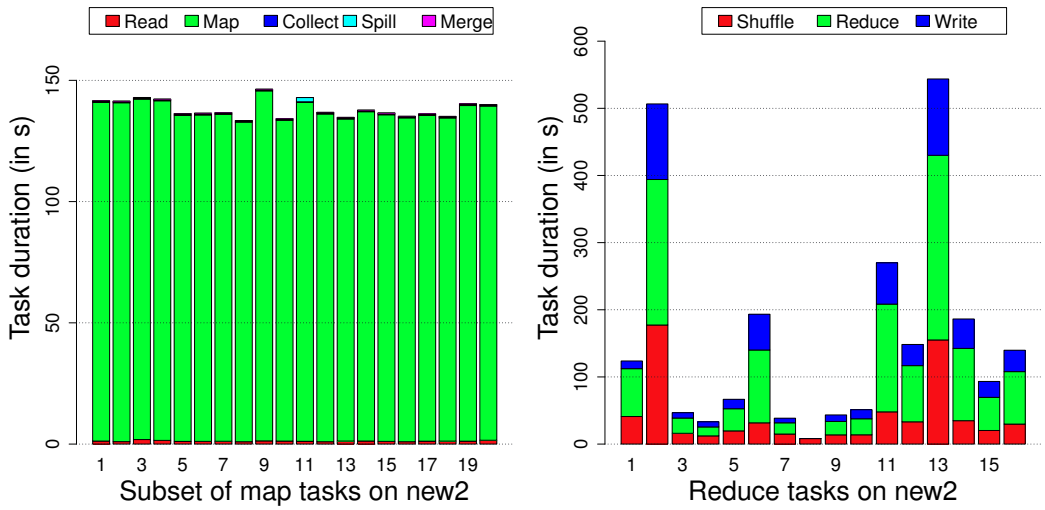


Figure 6.14: Job trace of KMeans on *new2* platform.

6.7 Summary

Many existing Hadoop clusters are empirically tuned for optimized performance for production workloads. Often, the jobs are partitioned into different classes of service and then processed by different Hadoop clusters with specially created management and resource allocation strategies to guarantee performance isolation and predictable completion time. When the time for hardware upgrades comes, system administrators have a long list of challenges and performance questions to answer. They are responsible for selecting a well-performing platform for a new Hadoop cluster, adequately size it, and obtain it at a sensible and justified price/performance ratio. To assist system administrators in solving these problems we introduce ARIEL – a novel framework that enables performance assessment and comparison of different hardware choices as candidates for upgrading existing MapReduce clusters. Our approach aims to eliminate guess-based manual processes and offers an automated solution. We envision that the proposed approach might be of interest to hardware and service providers. Acquiring a Hadoop cluster might become easier if the providers can offer an additional comparison of different hardware choices and their impact on performance of MapReduce processing pipeline.

Chapter 7

Related Work

In this chapter, we provide an overview of the state of the art in scheduling, performance modeling, and simulating MapReduce environments and contrast our work.

7.1 MapReduce Scheduling

The scheduling of incoming jobs and the assignment of processors to the scheduled jobs has been an important factor for optimizing the performance of parallel and distributed systems. It has been studied extensively in scheduling theory (see a variety of papers and textbooks on the topic [43, 44, 45, 46, 47, 48, 49, 50, 51]). Designing an efficient distributed server system often assumes choosing the best task assignment policy for a given model and user requirements. However, this question is still open for many models.

Originally, MapReduce was designed for periodically running large batch workloads. With a primary goal of minimizing the job makespan the simple *FIFO* scheduler was very efficient. As the number of users sharing the same MapReduce cluster increased, a new *Capacity* scheduler [27] was introduced to support more efficient cluster sharing. The capacity scheduler partitions the resources into pools and provides separate queues and priorities for each pool. However, users and system administrators do need to answer an additional question: how much resources do the time-sensitive jobs require and how to translate these requirements into the capacity scheduler settings? This question is still open: there are many research efforts discussed below that aim to design a MapReduce performance model for resource provisioning and predicting the job completion time.

In order to maintain fairness between different users, the *Hadoop Fair Scheduler* (HFS) [20] allocates equal shares to each of the users running the

MapReduce jobs. It also tries to maximize data locality by delaying the scheduling of the task, if no local data is available. Similar fairness and data locality goals are pursued in *Quincy* scheduler [52] proposed for the Dryad environment [7]. However, both HFS and Quincy do not provide any special support for achieving the application performance goals and service level objectives.

FLEX [28] extends HFS by proposing a special slot allocation schema that aims to optimize explicitly some given scheduling metric. FLEX relies on the speedup function of the job (for map and reduce stages) that produces the job execution time as a function of the allocated slots. FLEX does not provide a technique for job profiling or a detailed MapReduce performance model, but instead uses a set of simplifying assumptions about the job execution, tasks durations and job progress over time.

Another interesting extension of the existing Hadoop FIFO and fair-share schedulers using the dynamic proportional sharing mechanism is proposed in [29]. The new Dynamic Priority (DP) scheduler allows users to purchase and bid for capacity (map and reduce slots) dynamically by adjusting their spending over time. While this approach allows dynamically controlled resource allocation, it is driven by economic mechanisms rather than a performance model and/or application profiling.

Moseley et al. [53] formalize MapReduce scheduling as a generalization of the classical two-stage flexible flow-shop problem with identical machines. They provide approximate algorithms for minimizing the makespan of MapReduce jobs in offline and online scenarios.

7.2 MapReduce Performance Modeling

Polo et al. [54] introduce an online job completion time estimator which can be used for adjusting the resource allocations of different jobs. However, their estimator tracks the progress of the map stage alone and has no information or control over the reduce stage. Ganapathi et al. [55] use Kernel Canonical Correlation Analysis to predict the performance of MapReduce workloads. However, they concentrate on Hive queries and do not attempt to model the actual execution of the MapReduce job. The authors discover the feature vectors through statistical correlation.

Morton et al. [56] propose *ParaTimer* for estimating the progress of parallel queries expressed as Pig scripts [14] that translate into directed acyclic graphs (DAGs) of MapReduce jobs. In their earlier work [57], they designed *Parallax* – a progress estimator that aims to predict the completion time of a limited class of Pig queries that translate into a sequence of MapReduce jobs. In both papers, instead of using a detailed profiling technique similar to our work, the authors rely on previous debug runs of the same query for estimating the throughput of map and reduce stages on the input data samples provided by the user.

Phan et al. [58] aim to build an optimal schedule for a set of MapReduce jobs with given deadlines. The authors investigate different factors that impact job performance and its completion time such as the ratio of slots to core, the number of concurrent jobs, data placement, etc. MapReduce jobs with a single map and reduce waves are considered, and the scheduling problem is formulated as a constraint satisfaction problem (CSP). These assumptions and the CSP complexity issues make it difficult to generalize this approach.

Cardosa at al. [59] propose a provisioning framework, called STEAMEngine, which is a family of provisioning algorithms to optimize different user or provider metrics, such as runtime, cost, throughput, or energy. STEAMEngine accumulates the database of historic observations (the same job completion times for different input dataset sizes and cluster sizes). The authors suggest performing a few experiments with a small dataset and different cluster size combinations to enable the extrapolation technique. The profile database has separate runtimes for the map and reduce phases of these job executions.

Originally, Hadoop was designed for homogeneous environments. There has been recent interest [60] in heterogeneous MapReduce environments. Our approach and the proposed scaling technique will efficiently work in heterogeneous MapReduce environments. In a heterogeneous cluster, the slower nodes would be reflected in the longer tasks durations, and they all would contribute to the average and maximum task durations in the job profile. While we do not explicitly consider different types of nodes, their performance is reflected in the job profile and used in the future predictions.

Much of the recent work also focuses on anomaly detection, stragglers and outliers control in MapReduce environments [60, 61, 62, 63] as well as op-

timization and tuning cluster parameters and testbed configuration [64, 65]. While this work is orthogonal to our research, the results are important for performance modeling in MapReduce environments. Providing more reliable, well performing, balanced environment enables reproducible results, consistent job executions and supports more accurate performance modeling and predictions.

7.3 MapReduce Simulators

While MapReduce is a relatively new programming paradigm, there are a few on-going efforts on developing simulation tools for MapReduce environments.

The designers of *MRPerf* [24] aim to provide a fine-grained simulation of MapReduce setups throughout different phases. To model inter- and intra rack task communications over network as well as to accurately model the network behavior, *MRPerf* is based on the widely-used ns-2 network simulator [66]. The authors are interested in modeling different cluster topologies and their impact on the MapReduce job performance. For map/reduce task modeling, *MRPerf* creates a number of simulated nodes, where each node might have several processors and a single disk (it is the *MRPerf* limitation). There are a few simplifying assumptions about the application behavior: that a job has simple map and reduce tasks with compute time requirements that are proportional to the data size but not the actual content of the data.

In our work, we focus on simulating the job master decisions and the task/slot allocations across multiple jobs. We do not simulate details of the TaskTrackers (their hard disks or network packet transfers) as done by *MRPerf*. In spite of this, our approach accurately reflects the job processing because our profiling technique represents job latencies during the different phases of MapReduce processing in the cluster. Our approach does not have many of *MRPerf*'s limitations. Moreover, it is very fast compared to *MRPerf* which deals with network-packet level simulations.

Another effort presents a simulator [67] that utilizes SimJava [68] and GridSim [69]. This tool is in very early stages of development. In the short paper, authors describe their goals to build a simulator for assessing a future application design (i.e., applications that do not yet exist) rather than replaying traces of already existing applications. The authors are interested is

evaluating the application scalability and parameter/configuration tuning.

Cardona et al. [70] discuss how to build a federated MapReduce environment on top of different Hadoop clusters. There are quite a few issues that need to be reconsidered in Hadoop while building such a system. One of the issues is that the original Hadoop assumes a homogeneous environment, and there are a few internal mechanisms that are grounded on this assumption. The authors discuss the modifications to Hadoop that are useful to support heterogeneity. To justify the set of proposed modifications the authors design a simulation environment based on GridSim [69].

The closest approach to our SimMR is Apache's MapReduce simulator, called Mumak [31]. This simulator replays traces collected with a log processing tool, called Rumen [32]. The main difference between Mumak and SimMR is that Mumak omits modeling the shuffle/sort phase. As we have shown this omission could lead to inaccurate results. We believe that our modeling approach embodied in SimMR could be adopted by Mumak.

7.4 Comparison of Hardware Alternatives

The problem of predicting the application performance on a new or different hardware has been an open challenge for a long time. The body of work [71, 72] that is closely related to our approach dates back almost two decades. In 1995, Larry McVoy and Carl Staelin introduced the *lmbench* [71] – a suite of operating system microbenchmarks that provides an extensible set of portable programs for system profiling and the use in cross-platform comparisons. Each microbenchmark captures some unique performance properties and features that were present in popular and important applications of that time. In their later work [73], they continue extending the *lmbench* suite with additional useful benchmarks, such as *mhz*, that is a portable ANSI/C program for determining the processor clock speed in a platform independent fashion: it does not depend on any specific compiler, operating system, or processor. Although such microbenchmarks can be useful in understanding the end-to-end behavior of a system, the results of these microbenchmarks provide little information to indicate how well a particular application will perform on a particular system.

Seltzer et. al. [72] argue for an application-specific approach to bench-

marking. The authors suggest a vector-based approach for characterizing an underlying system by a set of microbenchmarks (e.g., *lmbench*) that describes the behavior of the fundamental primitives of the system. The results of these microbenchmarks constitute the *system* vector and it characterizes the system performance. They construct an *application* vector that quantifies the way the application makes use of the various primitives supported by the system. The product of these two vectors yields a relevant performance metric. We adopt a similar approach in our design: we use a set of specially generated microbenchmarks to characterize the pair of underlying Hadoop clusters: *old* and *new* ones. Then we apply the derived model (the analogy to a *system* vector) to the application traces (the analogy to the *application* vector) and use it for predicting the application performance on a *new* Hadoop cluster. However in our work, we derive a *relative model* that predicts performance differences between a pair of given Hadoop clusters. This contrasts with an *absolute model* that is applied directly to the analyzed workload or application for predicting its performance.

Recently, several *absolute models* have been designed for predicting the performance of MapReduce applications [74, 39, 75, 30]. Tian and Chen [75] propose an approach to predict MapReduce program performance from a set of test runs on small input datasets and small number of nodes. By executing 25-60 diverse test runs the authors create a training set for building a regression-based model of a given application. The derived model is able to predict the application performance on a larger input and a different size Hadoop cluster.

Starfish [39] applies dynamic Java instrumentation to collect a run-time monitoring information about job execution at a fine granularity and by extracting a diverse variety of metrics. Such a detailed job profiling enables the authors to predict job execution under different Hadoop configuration parameters, and automatically derive an optimized configuration. However, collecting a large set of metrics comes at a cost, and to avoid significant overhead profiling should be applied to a small fraction of tasks. The authors introduce analytic and simulation models for predicting the job completion time and addressing a cluster sizing problem [74].

Prior examples of successfully building relative models include a *relative fitness model* for storage devices [76] using CART models, and a *relative model* between the native and virtualized systems [77] based on a linear-

regression technique. The main challenge outlined in both works [76, 77] is the benchmark coverage: if a benchmark collection used for system profiling is not representative or complete to reflect important workload properties then the created model might be inaccurate. Herodotou et. al. [74] attempt to derive a *relative model* for Hadoop clusters comprised of different Amazon EC2 instances. They use the Starfish profiling technique and a small set of six benchmarks to exercize job processing with data compression and combiner turned on and off. The model is generated with the M5 Tree Model approach [78]. The authors report that the predicted makespan is within 15% of the measured one for the *combined* execution of six applications. It is hard to directly compare their approach to ours due to insufficient details. It is unclear whether the model derived using six benchmarks with fixed parameters can accurately predict performance of general MapReduce applications on diverse hardware. In our evaluation study, we observe that fixed parameters of map and reduce selectivities in benchmarks lead to inaccurate results. We justify our decision choices for microbenchmarks suite and derived model via a detailed performance study.

7.5 Performance Modeling of other Computing Systems

Lilja [79] and Landberg [80] provide an excellent background of the fundamental techniques used in analyzing and understanding the performance of computer system. Performance models have been built for numerous computing systems like hard disks [81], TCP [82] and high performance computing [83] systems.

Chapter 8

Conclusion

We have detailed work that supports our hypothesis, that *performance modeling of MapReduce environments through a combination of measurement, simulation, and analytical modeling for enabling different service level objectives is feasible, novel, and useful.* Our performance modeling tools and its applications have led to several research publications [30, 84, 85, 86, 87]. Our final work looks at building robust models for capacity planning and comparing different hardware alternatives (Chapter 6). Taken as a whole, our research demonstrates practical performance models and tools that enable different service level objectives in MapReduce environments.

References

[1] Techcrunch, http://goo.gl/jWWH8. (page 1)

[2] http://www.informationweek.com/news/software/bi/207800705. (page 1)

[3] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, "Data warehousing and analytics infrastructure at facebook," in *Proceedings of the 2010 International Conference on Management of Data*. ACM, pp. 1013–1020. (page 1)

[4] http://www.archive.org/about/faqs.php. (page 1)

[5] D. D., "Examples of future large scale scientific databases," in *Extremely Large Databases Workshop*, 2007. (page 1)

[6] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008. (page 1), (page 4), (page 7), (page 9)

[7] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: distributed data-parallel programs from sequential building blocks," *ACM SIGOPS OS Review*, vol. 41, no. 3, p. 72, 2007. (page 1), (page 3), (page 117)

[8] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. Gunda, and J. Currey, "Dryadlinq: A system for general-purpose distributed data-parallel computing using a high-level language," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 1–14. (page 1)

[9] M. Zaharia, M. Chowdhury, M. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*, 2010, pp. 1–10. (page 1)

[10] "Powered by Hadoop," http://wiki.apache.org/hadoop/PoweredBy. (page 1)

[11] D. Borthakur, K. Muthukkaruppan, K. Ranganathan, S. Rash, J. Sarma, N. Spiegelberg, D. Molkov, R. Schmidt, J. Gray, H. Kuang et al., "Apache hadoop goes realtime at facebook," in *Proceedings of the 2011 international conference on Management of data*. ACM, 2011, pp. 1071–1080. (page 2)

[12] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. Rao, V. Sankarasubramanian, S. Seth et al., "Nova: continuous pig/hadoop workflows," in *Proceedings of the 2011 international conference on Management of data*. ACM, 2011, pp. 1081–1090. (page 2)

[13] http://perspectives.mvdirona.com/2008/11/28/CostOfPowerInLargeScaleDataCenters.aspx. (page 2)

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *Proceedings of SIGMOD*. ACM, 2008, pp. 1099–1110. (page 3), (page 118)

[15] A. Thusoo, J. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *Proceedings of the VLDB Endowment*, vol. 2, no. 2, pp. 1626–1629, 2009. (page 3)

[16] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," in *Proceedings of the ACM SIGCOMM 2008 conference on Data communication*. ACM, 2008, pp. 63–74. (page 4)

[17] A. Greenberg, J. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. Maltz, P. Patel, and S. Sengupta, "Vl2: a scalable and flexible data center network," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 51–62, 2009. (page 4)

[18] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu, "Bcube: a high performance, server-centric network architecture for modular data centers," *ACM SIGCOMM Computer Communication Review*, vol. 39, no. 4, pp. 63–74, 2009. (page 4)

[19] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001. (page 8)

[20] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: A simple technique for achieving locality and fairness in cluster scheduling," in *Proc. of EuroSys*. ACM, 2010, pp. 265–278. (page 10), (page 45), (page 46), (page 56), (page 61), (page 116)

[21] O. OMalley and A. Murthy, "Winning a 60 second dash with a yellow elephant," 2009. (page 11)

[22] R. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Tech. Journal*, vol. 45, no. 9, pp. 1563–1581, 1966. (page 15)

[23] H. Kwak, C. Lee, H. Park, and S. Moon, "What is Twitter, a social network or a news media?" in *Proc. of intl. conference on World Wide Web*. ACM, 2010, pp. 591–600. (page 32), (page 53)

[24] G. Wang, A. Butt, P. Pandey, and K. Gupta, "A simulation approach to evaluating design decisions in mapreduce setups," in *Intl. Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, 2009. (page 41), (page 48), (page 119)

[25] S. Kavulya, J. Tan, R. Gandhi, and P. Narasimhan, "An Analysis of Traces from a Production MapReduce Cluster," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*. IEEE, 2010, pp. 94–103. (page 41), (page 42), (page 80)

[26] A. Thusoo and et al., "Data warehousing and analytics infrastructure at Facebook," in *Proc. of the Intl. Conference on Management of Data*. ACM, 2010. (page 46)

[27] Apache, "Capacity Scheduler Guide," 2010. [Online]. Available: http://hadoop.apache.org/common/docs/r0.20.1/capacity_scheduler.html (page 46), (page 56), (page 80), (page 116)

[28] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, and A. Balmin, "FLEX: A Slot Allocation Scheduling Optimizer for MapReduce Workloads," *To appear in Middleware*, 2010. (page 46), (page 117)

[29] T. Sandholm and K. Lai, "Dynamic Proportional Share Scheduling in Hadoop," *LNCS: Proc. of the 15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010. (page 46), (page 117)

[30] A. Verma, L. Cherkasova, and R. Campbell, "ARIA: automatic resource inference and allocation for mapreduce environments," in *Proc. of IEEE/ACM Intl. Conference on Autonomic Computing (ICAC)*, 2011. (page 46), (page 57), (page 58), (page 87), (page 107), (page 113), (page 121), (page 123)

[31] Apache, "Mumak: Map-Reduce Simulator. https://issues.apache.org/jira/browse/MAPREDUCE-751." (page 47), (page 52), (page 89), (page 120)

[32] Apache, "Rumen: a tool to extract job characterization data from job tracker logs. https://issues.apache.org/jira/browse/MAPREDUCE-728." (page 47), (page 52), (page 89), (page 120)

[33] X. Wang, C. Olston, A. Sarma, and R. Burns, "CoScan: Cooperative Scan Sharing in the Cloud," in *Proc. of SOCC*, 2011. (page 69)

[34] S. Johnson, "Optimal two-and three-stage production schedules with setup times included," *Naval research logistics quarterly*, vol. 1, no. 1, pp. 61–68, 1954. (page 70), (page 73)

[35] A. Verma, L. Cherkasova, and R. Campbell, "Play It Again, SimMR!" in *Proc. of IEEE Intl. Conference on Cluster Computing*, 2011. (page 76), (page 79), (page 81), (page 87), (page 89), (page 108), (page 114)

[36] M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*. WH Freeman & Co., 1979. (page 78)

[37] Apache, "GridMix http://hadoop.apache.org/mapreduce/docs/current/gridmix.html." (page 85)

[38] Apache, "Hadoop: TeraGen Class. http://hadoop.apache.org/common/docs/r0.20.2/api/org/apache/hadoop/examples/terasort/TeraGen.html." (page 93)

[39] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. Cetin, and S. Babu, "Starfish: A self-tuning system for big data analytics," in *Proc. of Conf. on Innovative Data Systems Research (CIDR)*, 2011. (page 98), (page 121)

[40] "BTrace: A Dynamic Instrumentation Tool for Java. http://kenai.com/projects/btrace." (page 98)

[41] P. Holland and R. Welsch, "Robust regression using iteratively reweighted least-squares," *Communications in Statistics-Theory and Methods*, vol. 6, no. 9, pp. 813–827, 1977. (page 100)

[42] F. Ahmad, S. Chakradhar, A. Raghunathan, and T. Vijaykumar, "Tarazu: Optimizing MapReduce on heterogeneous clusters," in *Proc. of ASPLOS*, 2012. (page 101)

[43] J. Blazewicz, *Scheduling in computer and manufacturing systems*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1996. (page 116)

[44] G. Blelloch, P. Gibbons, and Y. Matias, "Provably efficient scheduling for languages with fine-grained parallelism," *Journal of the ACM (JACM)*, vol. 46, no. 2, pp. 281–321, 1999. (page 116)

[45] R. Blumofe and C. Leiserson, "Scheduling multithreaded computations by work stealing," *Journal of the ACM (JACM)*, vol. 46, no. 5, pp. 720–748, 1999. (page 116)

[46] C. Chekuri and M. Bender, "An efficient approximation algorithm for minimizing makespan on uniformly related machines," *Integer Programming and Combinatorial Optimization*, pp. 383–393, 1998. (page 116)

[47] C. Chekuri and S. Khanna, "Approximation algorithms for minimizing average weighted completion time," *Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press, Inc., Boca Raton, FL, USA*, 2004. (page 116)

[48] B. Lampson, "A scheduling philosophy for multiprocessing systems," *Communications of the ACM*, vol. 11, no. 5, p. 360, 1968. (page 116)

[49] J. Leung, *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004. (page 116)

[50] M. Pinedo, *Scheduling: theory, algorithms, and systems*. Springer Verlag, 2008. (page 116)

[51] L. Tan and Z. Tari, "Dynamic task assignment in server farms: Better performance by task grouping," in *Computers and Communications, 2002. Proceedings. ISCC 2002. Seventh International Symposium on*. IEEE, 2002, pp. 175–180. (page 116)

[52] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: Fair scheduling for distributed computing clusters," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. Citeseer, 2009, pp. 261–276. (page 117)

[53] B. Moseley, A. Dasgupta, R. Kumar, and T. Sarlós, "On scheduling in map-reduce and flow-shops," in *Proceedings of SPAA*, 2011. (page 117)

[54] J. Polo, D. Carrera, Y. Becerra, J. Torres, E. Ayguadé, M. Steinder, and I. Whalley, "Performance-driven task co-scheduling for mapreduce environments," in *12th IEEE/IFIP Network Operations and Management Symposium*. ACM, 2010. (page 117)

[55] A. Ganapathi, Y. Chen, A. Fox, R. Katz, and D. Patterson, "Statistics-driven workload modeling for the cloud," in *Proceedings of 5th International Workshop on Self Managing Database Systems (SMDB)*, 2010. (page 117)

[56] K. Morton, M. Balazinska, and D. Grossman, "ParaTimer: a progress indicator for MapReduce DAGs," in *Proceedings of SIGMOD*. ACM, 2010, pp. 507–518. (page 118)

[57] K. Morton, A. Friesen, M. Balazinska, and D. Grossman, "Estimating the progress of MapReduce pipelines," in *Proceedings of ICDE*. IEEE, 2010, pp. 681–684. (page 118)

[58] L. Phan, Z. Zhang, B. Loo, and I. Lee, "Real-time MapReduce Scheduling," in *Technical Report No. MS-CIS-10-32, University of Pennsylvania*, 2010. (page 118)

[59] M. Cardosa, P. Narang, A. Chandra, H. Pucha, and A. Singh, "Driving MapReduce Provisioning in the Cloud," in *Technical Report TR10-023, University of Minnesota*, 2010. (page 118)

[60] M. Zaharia, A. Konwinski, A. Joseph, R. Katz, and I. Stoica, "Improving mapreduce performance in heterogeneous environments," in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, 2008, pp. 29–42. (page 118)

[61] J. Tan, X. Pan, E. Marinelli, S. Kavulya, R. Gandhi, and P. Narasimhan, "Kahuna: Problem diagnosis for mapreduce-based cloud computing environments," in *Network Operations and Management Symposium (NOMS)*. IEEE, 2010, pp. 112–119. (page 118)

[62] J. Tan, S. Kavulya, R. Gandhi, and P. Narasimhan, "Visual, log-based causal tracing for performance debugging of mapreduce systems," in *2010 International Conference on Distributed Computing Systems*. IEEE, 2010, pp. 795–806. (page 118)

[63] G. Ananthanarayanan, S. Kandula, A. Greenberg, I. Stoica, Y. Lu, B. Saha, and E. Harris, "Reining in the outliers in map-reduce clusters using mantri," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–16. (page 118)

[64] Intel, "Optimizing Hadoop* Deployments," http://communities.intel.com/docs/DOC-4218, 2010. (page 118)

[65] K. Kambatla, A. Pathak, and H. Pucha, "Towards optimizing hadoop provisioning in the cloud," in *Proc. of the First Workshop on Hot Topics in Cloud Computing*, 2009. (page 118)

[66] "Network Simulator (NS2)," http://www.isi.edu/nsnam/ns, 2005. (page 119)

[67] S. Hammoud, M. Li, Y. Liu, N. Alham, and Z. Liu, "MRSim: A Discrete Event Based MapReduce Simulator," in *Fuzzy Systems and Knowledge Discovery (FSKD), 2010 Seventh International Conference on*, vol. 6, 2010. (page 119)

[68] F. Howell and R. McNab, "SimJava: A Discrete Event Simulation Library for Java," *Simulation Series*, vol. 30, 1998. (page 119)

[69] R. Buyya and M. Murshed, "Gridsim: A Toolkit for Modeling and Simulation of Distributed Resource Management and Scheduling for Grid Computing," *Concurrency and Computation: Practice and Experience*, vol. 14, no. 13-15, 2002. (page 119), (page 120)

[70] K. Cardona, J. Secretan, M. Georgiopoulos, and G. Anagnostopoulos, "A grid based system for data mining using MapReduce," TR-2007-02, AMALTHEA, Tech. Rep., 2007. (page 120)

[71] L. McVoy and C. Staelin, "lmbench: Portable tools for performance analysis," in *Proc. of USENIX ATC*, 1996. (page 120)

[72] M. Seltzer, D. Krinsky, K. Smith, and X. Zhang, "The case for application-specific benchmarking," in *Proc. of Workshop on Hot Topics in Operating Systems*, 1999. (page 120)

[73] C. Staelin and L. McVoy, *mhz: Anatomy of a micro-benchmark*. Hewlett Packard Laboratories, 1997. (page 120)

[74] H. Herodotou, F. Dong, and S. Babu, "No one (cluster) size fits all: Automatic cluster sizing for data-intensive analytics," in *Proc. of ACM Symposium on Cloud Computing*, 2011. (page 121), (page 122)

[75] F. Tian and K. Chen, "Towards optimal resource provisioning for running MapReduce programs in public clouds," in *Proc. of IEEE Intl. Conference on Cloud Computing*, 2011. (page 121)

[76] M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger, "Modeling the relative fitness of storage," in *Proc. of ACM SIGMETRICS*, 2007. (page 121)

[77] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy, "Profiling and modeling resource usage of virtualized applications," in *Proc. of ACM/IFIP/USENIX Middleware*, 2008. (page 121)

[78] J. Quinlan, "Learning with continuous classes," in *Proc. of Australian joint Conference on Artificial Intelligence*, 1992. (page 122)

[79] D. Lilja, *Measuring computer performance: a practitioner's guide*. Cambridge University Press, 2005. (page 122)

[80] S. Lavenberg, *Computer Performance Modeling Handbook*. Orlando, FL, USA: Academic Press, Inc., 1983. (page 122)

[81] Q. Zhu, Z. Chen, L. Tan, Y. Zhou, K. Keeton, and J. Wilkes, "Hibernator: helping disk arrays sleep through the winter," *SIGOPS Operating Systems Review*, vol. 39, pp. 177–190, 2005. (page 122)

[82] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: a simple model and its empirical validation," *SIGCOMM Commun. Rev.*, vol. 28, pp. 303–314, 1998. (page 122)

[83] K. Datta, S. Kamil, S. Williams, L. Oliker, J. Shalf, and K. Yelick, "Optimization and performance modeling of stencil computations on modern microprocessors," *SIAM review*, vol. 51, no. 1, pp. 129–159, 2009. (page 122)

[84] A. Verma, L. Cherkasova, and R. Campbell, "SLO-Driven Right-Sizing and Resource Provisioning of MapReduce Jobs," in *Proceedings of Workshop on Large Scale Distributed Systems and Middleware (LADIS) in conjunction with VLDB*, 2011. (page 123)

[85] A. Verma, L. Cherkasova, and R. Campbell, "Resource Provisioning Framework for MapReduce Jobs with Performance Goals," in *Proceedings of ACM/IFIP/USENIX International Middleware Conference*, 2011. (page 123)

[86] A. Verma, L. Cherkasova, V. S. Kumar, and R. Campbell, "Deadline-based Workload Management for MapReduce Environments: Pieces of the Performance Puzzle," in *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, 2012. (page 123)

[87] A. Verma, L. Cherkasova, and R. Campbell, "Play it again, SimMR!" in *Proceedings of IEEE International Conference on Cluster Computing*, 2011. (page 123)